Saarland University
Faculty of Natural Science and Technology I
Department of Computer Science

Master's thesis

CDCL with Reduction

submitted by

Andreas Teucke

submitted

May 24, 2013

Supervisor

Prof. Dr. Christoph Weidenbach

Reviewers

Christoph Weidenbach

Sebastian Hack

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,

Date	Signature

Abstract

Conflict Driven Clause Learning (CDCL) [1] is a very successful algorithm for solving propositional Satisfiability. At its core lies unit propagation which forces the value of a propositional variable when it is contained in a unit clause. This thesis extends CDCL to RCDCL (Reduction Conflict Driven Clause Learning) where unit propagation is generalized to subsumption resolution and thereby increases the power of propagation. Intuition suggests that this will decrease the depth of proof search.

I define the rules of the extended calculus, prove correctness, and describe their implementation.

Acknowledgment

My deepest gratitude is due to Professor Dr. Christoph Weidenbach for offering me this topic. Our meetings were interesting, and I was looking forward to them each week. I also thank Dr. Uwe Waldmann for his lectures introducing me to Automated Reasoning. Further Armin Biere's lecture on SAT during Summer school and his solver cleaneling greatly aided me in the implementation of my own solver. Finally, I am deeply grateful to my family for their support and reminding me to work when I was too lazy and to take breaks when I was working too much.

Contents

1 Introduction **11**
1.1 Structure of the Thesis . . . 12

2 Preliminaries **13**
2.1 Notations . . . 13
2.2 CDCL Calculus . . . 13
2.2.1 Example . . . 16
2.3 Subsumption and Subsumption Resolution . . . 16

3 The RCDCL Calculus **19**
3.1 States . . . 19
3.2 Semantics . . . 20
3.3 RCDCL Rules . . . 21
3.3.1 The Propagation Phase . . . 21
3.3.2 The Conflict Phase . . . 22
3.4 Non-linear Conflict Analysis . . . 24

4 Properties of RCDCL **27**
4.1 Consistency of the Model Assumption . . . 27
4.2 Consistency of the Conflict Clauses . . . 29
4.3 Correctness of the Conflict Analysis . . . 33
4.4 Model Construction . . . 36
4.5 Soundness . . . 39
4.6 Redundancy of Learnt Clauses . . . 41
4.7 Termination . . . 42

5 Implementation **47**
5.1 Data Structures . . . 47
5.2 Propagation Phase . . . 48
5.2.1 Propagation . . . 48
5.2.2 Backward Subsumption Resolution . . . 49
5.2.3 Determining a Literal's Truth value . . . 51

5.2.4 Partner Selection for Subsumption Resolution 51
5.2.5 The *LitSet* Data Structure 53
5.2.6 Propagation after Restart 54
5.3 Conflict Phase 56
5.3.1 Conflict Analysis 56
5.3.2 Forward Reduction of Learned Clauses 58

6 Results 61
6.1 The Effect of Subsumption on Satisfied Clauses 61
6.2 Pure Literal Elimination 62
6.3 Development 62
6.4 Evaluation 64
6.4.1 First Experiment 64
6.4.2 Second Experiment 65
6.5 Conclusion 67
6.5.1 Future Work 67

Bibliography 69

1 Introduction

Since the introduction of the efficient 2-watched literal data structure [10] and conflict analysis using the implication graph [12], Boolean constraint propagation (BCP) is at the core of any modern conflict driven clause learning (CDCL) based SAT solver [12] [6]. From delaying costly splitting decisions in the search to providing the implication graph; the quality of a solver depends on the efficiency and power of BCP. However the propagated information comes from only one restricted source: Unit clauses under the partial assignment. Intuition suggests that a generalization of BCP that uses more information about the state would be more powerful and thereby, could shorten the search.

As a generalization, I propose an established method from pre-processing called Subsumption resolution [5] - also known under similar names such as matching replacement or decremental resolution [14].

$$\frac{C \lor L \qquad D \lor \neg L}{C \lor L \qquad D} \qquad C \subseteq D$$

In the case that the resolvent of two clauses subsumes one of its parents the subsumed clause is replaced. This can also be interpreted as deleting the resolution literal from the parent clause. The special case that C is empty is equivalent to BCP.

To reasonably use Subsumption resolution under a partial assignment as part of BCP, important general properties of CDCL implementations need to be preserved. For example, efficient backtracking without requiring updates of the clause data structures and the ability to analyze conflicts with an implicit conflict graph.

To solve both problems, I modify the assignment stack to also hold reduction literals. Whereas propagated literals are associated with their asserting clause and denote the forced assignment of the unit literal, reduction literals are linked with both parent clauses of a Subsumption resolution and denote the removal of the literal from the affected parent clause.

I extend a standard CDCL calculus with several new rules to handle propagation, conflict analysis, and backtracking of reduction literals. The new

calculus is called reduction conflict driven clause learning (RCDCL). Due to the added complexity in the model assumption and conflict clauses, conflict analysis involving Subsumption resolution can no longer be translated into a linear sequence of resolution steps as in CDCL. Instead, the analysis now involves branching resolution trees.

These changes to the definitions and behavior of the calculus prevent existing proofs for CDCL to be applied to RCDCL. Hence, I formulate and prove many new, strengthened lemmas and eventually, use those lemmas to also prove soundness and termination of RCDCL.

I have implemented the RCDCL calculus based on the CDCL solver SPASS-SATT [11]. There are two main tasks to complete.

The first is to find a method to identify Subsumption resolutions. As this involves comparing pairs of clauses, the complexity is inherently quadratic. Much care and specialized data structures are necessary to prevent this from completely dominating the solvers run time.

Secondly, in the abstract calculus the analysis of a conflict is now even exponential. However, I will show that a linear time implementation is still possible.

Lastly, I tested my implementation against itself without Subsumption resolution on several unsatisfiable benchmarks to discover the empirical effect of Subsumption resolution on the search.

1.1 Structure of the Thesis

In the following thesis, I will begin in chapter 2 by repeating the standard notations of SAT and explaining the underlying basic CDCL calculus. This calculus is then extended to RCDCL in three steps in chapter 3.

First, the modifications to the definitions and their semantics, then the new propagation rules, and last, the rules for conflict analysis.

In chapter 4, I prove correctness and termination of the new calculus. The implementation is described in chapter 5. The thesis is concluded in chapter 6 with insights during the development and the results of the implementation's evaluation.

2 Preliminaries

2.1 Notations

For an atom l, there are two literals, the positive literal l and the negative literal $\bar{l}$. A clause is a disjunction of literals, and a CNF formula is a conjunction of clauses. A clause can be seen as a finite set of literals and a CNF formula as a finite set of clauses. The length of a clause is the number of literals in the clause. A clause of length one is a unit clause.

A model for a CNF formula N is a function that maps literals in N to true or false, while a model assumption M is a partial function. Capital Letters $C, D, E \ldots$ denote clauses, $N, N', \ldots$ denote formulas, and $M, M_1, M_2, \ldots$ denote models and model assumptions.

If l is true, then $\bar{l}$ is false and vice versa. A literal l that does not have an assigned value is undefined. A clause C is satisfied by a model or model assumption if l is true for some literal $l \in C$ and falsified by it if every $l \in C$ is false. A falsified clause is also called a conflict clause. A model satisfies N if it satisfies every clause in N. N is satisfiable if there exists a satisfying model and unsatisfiable otherwise. If two CNF formulas N and N' are satisfied by the same models, they are equivalent.

2.2 CDCL Calculus

A CDCL Calculus is given in Figure 2.1. It formalizes DPLL with Boolean constraint propagation, backtracking, analysis of conflicts, and clause learning.

States are triples of model assumptions M, a CNF formula N, and a third component, which is either $\top, \bot$ or a conflict clause. A model assumption M is written as a list of decision and propagation literals where each literal $l \in M$ is true. Both types correspond to their respective rules that add them to the model assumption. Decision literals are written as $l^\top$, and propagation literals are written l^C, where C is a clause that serves as justification. The start state is $(\epsilon; N; \top)$ with the empty model assumption ϵ. The final

Decide
$(M; N; \top) \Rightarrow_{CDCL} (Ml^{\top}; N; \top)$,
if l is undefined in M.

Propagate
$(M; N; \top) \Rightarrow_{CDCL} (Ml^{C \vee l}; N; \top)$,
if $C \vee l \in N$, $M \models \neg C$, and L is undefined in M.

Conflict
$(M; N; \top) \Rightarrow_{CDCL} (M; N; D)$,
if $D \in N$ and $M \models \neg D$.

Fail
$(M; N; C) \Rightarrow_{CDCL} (\epsilon; N; \bot)$, if no $l^{\top}$ occurs in M.

Backtrack
$(Ml; N; D) \Rightarrow_{CDCL} (M; N; D)$,
if $\bar{l}$ does not occur in D.

Explain
$(Ml^{C \vee l}; N; D \vee \bar{l}) \Rightarrow_{CDCL} (M; N; D \vee C)$

Continue
$(Ml^{\top}; N; D) \Rightarrow_{CDCL} (Ml^{D}; N \cup \{D\}; \top)$, if $\bar{l} \in D$.

Figure 2.1: The CDCL Calculus

states are $(M; N; \top)$ where M is a satisfying model of N and the failed state $(\epsilon; N; \bot)$ representing the unsatisfiability of N. All other states are intermediate states, where $(M; N; \top)$ is called a propagation state and $(M; N; D)$ a conflict state.

The propagation phase includes the Decide, Propagate, and Conflict rules. As in DPLL [3], the model assumption is extended with new literals as long as the model assumption does not falsify a clause. New literals are added to the model assumption in two ways. With the Decide rule an arbitrary undefined literal is chosen. This is always possible. However, it has the drawback that the calculus possibly needs to return to this state and repeat with the negation of the literal. Hence, in general the worst case run time is exponential.

The aim of Boolean constraint propagation (BCP), included in the calculus with the Propagate rule, is to avoid decisions by instead adding literals whose

truth value is forced under the current state. If a clause is almost falsified, i.e., all its literals except one are false, whereas the last is still undefined, this literal is set to true. As otherwise the clause would be conflicting in any satisfying extension of the model assumption, this particular literal has only one possible assignment and ,therefore, can be added to the model assumption.

This continues until there are no more undefined literals or some clause in N is falsified. Then the Conflict rule applies; entering the conflict phase. The conflict phase consists of the Fail, Explain, Backtrack, and Continue rules.

The purpose of the conflict phase is now analyzing the conflict. It tries to find the earliest state to backtrack the model assumption to and creates a learnt clause. If there are no decisions in the model assumption, the Fail rule transitions into the fail state. As this means the current conflict is a direct consequence of N, N is unsatisfiable. Otherwise, the calculus uses the Explain or Backtrack rule to analyze the conflict depending on the top literal of the model assumption.

If the literal is not part of the conflict, i.e., assigns one of the conflict literals to false, the Backtrack rule simply removes it. This could be either a decision or propagation literal. As it does not affect the conflict, its truth value, and its appearance in the model assumption, is irrelevant. In the opposite case that the literal is relevant to the conflict the next step depends on whether the literal is a propagation or a decision. In the former case the conflict clause is resolved with the justification of the propagation literal, i.e., the clause that propagated the literal, and the result becomes the new conflict clause. The resolution of two clauses on an atom is their union minus the atom itself and any duplicates.

Repeating this step replaces the propagation literals in the conflict by the literals responsible for their propagation. More informally, backtracking the propagate steps in the model assumptions explains the conflict. As all literals in the conflict clause and the justification except for the propagation literal are falsified, the new conflict clause is again falsified. The purpose of this is to identify decisions that are not relevant to the conflict; therefore, exploring the second branch of those decision can be avoided.

The conflict analysis ends at the first relevant decision literal. At this point, the Continue rule adds the generated conflict clause to N as a new learnt clause. As it is still falsified, the clause is unit after backtracking the decision and therefore, immediately propagates the decision literal's negation. Furthermore, the resolvent is at each step implied by the two parent clauses, which are either a resolvent themselves or an element of N. Hence, the new clause is implied by N and therefore, does not change the satisfiability of N.

With this the conflict is resolved, and the calculus returns to the propagation phase to explore the second branch of the backtracked decision.

The calculus continues alternating between the two phases until either a satisfying model is found or the failed state is reached.

2.2.1 Example

In figure 2.2, I give an example, where the CDCL calculus is used to verify the unsatisfiability of the CNF formula $l_1 \vee l_2$, $l_1 \vee \bar{l}_2$, $\bar{l}_1 \vee l_2$, $\bar{l}_1 \vee \bar{l}_2$.

$$
\begin{array}{lllllll}
 & (& \varepsilon & ; N ; & \top &) & \text{start state} \\
\Rightarrow_{CDCL} & (& l_1^{\top} & ; N ; & \top &) & \text{Decide } l_1 \\
\Rightarrow_{CDCL} & (& l_1^{\top}\ l_2^{C} & ; N ; & \top &) & C \text{ propagates } l_2 \\
\Rightarrow_{CDCL} & (& l_1^{\top}\ l_2^{C} & ; N ; & \bar{l}_1 \vee \bar{l}_2 &) & D \text{ is a conflict clause} \\
\Rightarrow_{CDCL} & (& l_1^{\top} & ; N ; & \bar{l}_1 &) & C \text{ explains } \bar{l}_2 \\
\Rightarrow_{CDCL} & (& \bar{l}_1^{\bar{l}_1} & ; N \cup \{\bar{l}_1\} ; & \top &) & \text{Learn } \bar{l}_1 \text{ and continue} \\
\Rightarrow_{CDCL} & (& \bar{l}_1^{\bar{l}_1}\ l_2^{A} & ; N \cup \{\bar{l}_1\} ; & \top &) & A \text{ propagates } l_2 \\
\Rightarrow_{CDCL} & (& \bar{l}_1^{\bar{l}_1}\ l_2^{A} & ; N \cup \{\bar{l}_1\} ; & l_1 \vee \bar{l}_2 &) & B \text{ is a conflict clause} \\
\Rightarrow_{CDCL} & (& \varepsilon & ; N \cup \{\bar{l}_1\} ; & \bot &) & \text{Fail}
\end{array}
$$

Figure 2.2: Assume the CNF formula $N := A \wedge B \wedge C \wedge D$, where $A := l_1 \vee l_2$, $B := l_1 \vee \bar{l}_2$, $C := \bar{l}_1 \vee l_2$ and $D := \bar{l}_1 \vee \bar{l}_2$. The derivation shows one way for CDCL to reach the fail state in this case: As no other rule are applicable, the first step is Decide which is then followed by unit propagations that inevitably lead to two conflicts. Analyzing the first conflict inverts the initial decision, while the second conflict enables Fail.

2.3 Subsumption and Subsumption Resolution

Typically, CDCL solvers apply pre-processing in an attempt to simplify the CNF formula and to make subsequent solving faster.

Let C and D be two distinct clauses in a CNF formula N, and C is a subset of D. Then, any model M satisfying C automatically also satisfies D. If M falsifies D, it also falsifies C. Hence, N and the simplified $N \backslash \{D\}$ are equivalent. This is called Subsumption, and in this situation C subsumes D.

Now assume $C \vee l$ and $D \vee \bar{l}$ are two clauses in N, and again C is a subset of D. Then, their resolvent $C \vee D = D$ is a subset of $D \vee \bar{l}$. As adding a resolvent of two clauses in N preserves equivalence and the now added D

subsumes $D \vee \bar{l}$, the new CNF formula $N \setminus \{D \vee \bar{l}\} \cup \{D\}$ is equivalent to N. With this, the literal $\bar{l}$ was effectively removed. This is called Subsumption Resolution, and I will also refer to it as Reduction and say that $C \vee l$ reduces literal $\bar{l}$ in $D \vee \bar{l}$.

In the special case that C is empty and l is a unit clause, the condition $C \subseteq D$ holds for any clause $D \vee \bar{l}$. So the unit clause l reduces $\bar{l}$ in every clause in which $\bar{l}$ occurs, while it subsumes all clauses containing l. The result of Subsumption and Subsumption resolution combined are in this case the same as applying unit propagation.

Let us now consider again the CNF formula $l_1 \vee l_2$, $l_1 \vee \bar{l}_2$, $\bar{l}_1 \vee l_2$, $\bar{l}_1 \vee \bar{l}_2$. Due to lack of unit clauses, unit propagation can not be applied, and CDCL is forced to start with the decision rule. However, $l_1 \vee l_2$ reduces $\bar{l}_2$ in $l_1 \vee \bar{l}_2$, and the resulting clause l_1 subsumes both parent clauses. Continuing to apply Subsumption Resolution on $\bar{l}_1 \vee l_2$ and $\bar{l}_1 \vee \bar{l}_2$ and on the two reduced clauses l_1 and $\bar{l}_1$ results in the empty clause. Hence, we conclude unsatisfiability without ever applying the decision rule.

Therefore, Subsumption Resolution is a generalization of and more powerful than unit propagation.

3 The RCDCL Calculus

As I have just described the underlying CDCL calculus in the previous chapter, I will now explain how I extend it with Subsumption resolution to RCDCL.

3.1 States

Definition 3.1.1 (States). *A state $(M; N; s)$ is a triple consisting of the model assumption M, a set of propositional clauses N, and s which is $\top$, $\bot$ or a sequence of conflict clauses O.*

1. $(\epsilon; N; \top)$ *is the start state.*
2. $(\epsilon; N; \bot)$ *is the final state, where N is unsatisfiable.*
3. $(M; N; \top)$ *is an intermediate propagation state.*
4. $(M; N; O)$ *is an intermediate conflict state.*

Definition 3.1.2 (Model assumption M). *Let l be a literal, and C,D are clauses. The model assumption M is a sequence of literals of the following types:*

1. $l^{\top}$ *is a decision literal.*
2. l^{C} *is a propagation literal.*
3. l^{C}_{D} *is a reduced literal.*
4. S^{C}_{D} *is a subsumption.*

$M^p := \{l \mid l^C \in M \text{ or } l^{\top} \in M\}$ *is the propositional model of M.*

The superscript C is called the justification, whereas the subscript D denotes that something is specific to D. Hence, a reduced literal l^{C}_{D} describes that C reduces literal $\bar{l}$ in D. A subsumption S^{C}_{D} means that C subsumes D.

In a consistent Model assumption, both are uniquely defined by D. Therefore, C will be sometimes omitted. The decision and propagation literals have the same role as in standard CDCL.

Definition 3.1.3 (Conflict clauses)**.** *Let C and D be propositional clauses. A conflict clause is a set containing three types of conflict literals: Unmarked literals l, marked literals l_D, and chosen literals l^*_D, where a chosen literal is also a marked literal. Furthermore the following are defined:*

1. *$C_D := \{l_D \mid l \in C\}$ marks the literals of C.*
2. *$C_p := \{l \mid l \in C$ or $l_D \in C$ or $l^*_D \in C\}$ unmarks literals in C.*
3. *C^m and C^u are the sets of marked and unmarked literals in C.*

In CDCL, the conflict clause fulfills two essential invariants.
First, it is entailed by the set of clauses N, and second, it is false under the model assumption. In RCDCL, these invariants are generalized to lists of conflict clauses. However, the second invariant is then too strong to prove. Hence, it is weakened by requiring that only certain literals are false, namely, marked literals. Which those are is specified by the RCDCL rules.

Throughout the process of analyzing a conflict, each reduced literal l in a conflict clause needs to be resolved. This will require identifying which $\bar{l}^C_D$ in the Model assumption marks its reduction. Therefore, the origin clause D is used to mark conflict literals.

Chosen literals are the literals next in line to be resolved. While not directly essential to the calculus, they are treated separately in the proof for correctness and crucial for termination.

3.2 Semantics

Definition 3.2.1. *Let M be a model assumption and C, D clauses.*

1. *If $l^\top \in M$, $\bar{l}^\top \in M$, $l^C \in M$ or $\bar{l}^C \in M$, then l is defined in M.*
2. *If $\bar{l}_D \in M$, then the literal $l \in D$ is reduced under M.*
3. *If $\bar{l}_D \in M$, then the conflict literal l_D is reduced under M.*
4. *If $\bar{l}^\top \in M$, $\bar{l}^C \in M$ or $l \in D$ is reduced under M, then literal l is false in D under M.*

5. *If* $\bar{l}^{\top} \in M$*,* $\bar{l}^{C} \in M$ *or* l_D *is reduced under* M*,*
   then the conflict literal l_D *is false under* M*.*

As the second invariant now only requires all marked literals in a conflict clause to be false under M, I only define semantics for marked conflict literals. For unmarked literals only their presence in a conflict clause is relevant, but their truth value under M is not.

Definition 3.2.2 ($M(C)$)**.** *Let* M *be a model assumption and* C *a clause.* $M(C) := \{l \in C \mid l$ *is not false in* C *under* $M\}$*.*

$M(C)$ is how C would look if I used the subsumption resolution steps tracked by M to reduce C. As the model assumption marks reductions per clause, a literal's truth-value can no longer be determined globally as in CDCL. Instead, the truth-value of a literal also depends on the surrounding clause.

Definition 3.2.3 (Satisfiability)**.** *Let* M *be a model assumption, and* C, D *are clauses. A clause* D *is satisfied under* M*, denoted by* $M \models D$*, if* $S_D^C \in M$ *or* $l \in M_1(D)$*, where* $M = M_1 l^{\top} M_2$ *or* $M = M_1 l^C M_2$*.*

Informally, a clause is satisfied if it is either subsumed or a literal that has not been reduced is true. In Section 6.1 I will discuss subsumption and its influence on this definition.

3.3 RCDCL Rules

3.3.1 The Propagation Phase

The calculus begins in the start state $(\varepsilon; N; \top)$ in the Propagation phase, where it decides the truth value of literals and propagates resulting consequences. It stays in this phase until all literals have an assigned truth-value and a Model is found or at least one clause in N becomes false, when it leaves the phase with the Conflict rule.

Most rules in the propagation phase are the same as in CDCL with the only difference that their conditions are adapted to the changed notations. For example, instead of saying that a clause C is conflicting if it is false under M, we write $M(C) = \emptyset$, which denotes that all literals in C are either reduced or false.

Only additions are the Reduction and the Subsumption rules. The Subsumption Rule is not essential to the calculus. However, its condition is so close and even a bit simpler than the condition for Reduction that while

Unit Propagation
$(M; N \cup \{C\}; \top) \Rightarrow_{RCDCL} (Ml^C; N \cup \{C\}; \top)$,
where $M(C) = \{l\}$ and $M \not\models C$.

Reduction
$(M; N \cup \{C, D\}; \top) \Rightarrow_{RCDCL} (Ml^C_D; N \cup \{C, D\}; \top)$,
where $M(C) = C' \vee l$, $M(D) = D' \vee \bar{l}$, $C' \subseteq D'$, $M \not\models C$, and $M \not\models D$.

Subsumption
$(M; N \cup \{C, D\}; \top) \Rightarrow_{RCDCL} (MS^C_D; N \cup \{C, D\}; \top)$,
where $M(C) \subseteq M(D)$, $M \not\models C$, and $M \not\models D$.

Decide
$(M; N; \top) \Rightarrow_{RCDCL} (Ml^\top; N; \top)$,
where l is undefined in M.

Conflict
$(M; N \cup \{C\}; \top) \Rightarrow_{RCDCL} (M; N \cup \{C\}; C_C)$,
where $M(C) = \varnothing$.

Figure 3.1: The RCDCL Calculus - Propagation Phase

looking for reductions will inevitably also find subsumptions. Both have conditions as one would expect modulo the model assumption, but instead of changing the clause set they add a respective entry to the model assumption. This way backtracking undoes these operations quickly without affecting N.

3.3.2 The Conflict Phase

In the Conflict Phase, the calculus explicitly describes how conflicts and backtracking are handled and how the learnt clause is derived (Figure 3.2).

If there is a conflict but no decisions to backtrack, the calculus transitions into the fail state $(\varepsilon; N; \bot)$. Otherwise, it eventually returns to the propagation phase by adding a new clause to the CNF formula, which immediately forces a unit propagation.

The main difference compared to CDCL is the addition of the Explain Reduction Rule. Also, instead of explaining a propagation literal in a single step, the process is split into two parts: The Explain Propagation Rule, where the propagation literal's justification is added as a conflict clause, and the Resolution Rule, where the two top conflict clauses are replaced by their

Explain Propagation
$(Ml^{C\vee l}; N; D' \vee \bar{l}_D) \Rightarrow_{RCDCL} (Ml^{C\vee l}; N; D' \vee \bar{l}^*_D, C_{C\vee l} \vee l)$,
where no $l_E \in D' \vee \bar{l}_D$ is reduced under M.

Explain Reduction
$(M_1 l_D^{C\vee l} M_2; N; O, D' \vee \bar{l}_D) \Rightarrow_{RCDCL} (M_1 l_D^{C\vee l} M_2; N; O, D' \vee \bar{l}^*_D, F_{C\vee l} \vee G \vee l)$,
where $F := C \backslash M_1(D)$, $G := C \cap M_1(D)$, and no $l_E \in D'$ is reduced under M_1, i.e., $\bar{l}_D$ is the "left-most" reduced literal in the top conflict clause.

Resolution
$(M; N; O, D' \vee \bar{l}^*_D, C) \Rightarrow_{RCDCL} (M; N; O, D' \vee C^m)$,
where no $l_E \in C$ is reduced under M.

Backtrack1
$(Ml_D^{C'}; N; C) \Rightarrow_{RCDCL} (M; N; C)$,
where $\bar{l}_D \notin C$.

Backtrack2
$(Ml^{C'}; N; C) \Rightarrow_{RCDCL} (M; N; C)$,
where $\bar{l}_D \notin C$.

Backtrack3
$(Ml^{\top}; N; C) \Rightarrow_{RCDCL} (M; N; C)$,
where $\bar{l}_D \notin C$.

Backtrack4
$(MS^D_{C'}; N; C) \Rightarrow_{RCDCL} (M; N; C)$.

Continue
$(M\bar{l}^{\top}; N; C \vee l_D) \Rightarrow_{RCDCL} (Ml^{C_p \vee l}; N \cup \{C_p \vee l\}; \top)$,
where no $l_E \in C \vee l_D$ is reduced under M.

Fail
$(M; N; C) \Rightarrow_{RCDCL} (\epsilon; N; \bot)$, where no $l^{\top}$ occurs in M.

Figure 3.2: The RCDCL Calculus - Conflict Phase

resolvent. The Explain Reduction Rule has a similar effect to the Explain Propagation Rule as it adds a reduction's justification to the conflict clause list.

This approach is comparable to the way a stack machine evaluates arithmetic expression, where resolution is a binary operation and both Explain rules are load operations. Following this analogy, the extension to lists of conflict clauses instead of a single slot as in CDCL is necessary because resolution is no longer associative with the addition of reductions.

For the initial conflict clause, all reduced literals need to be resolved. Otherwise, the learnt clause might not be productive because the literals that are reduced in a conflict clause are not reduced under M in the learnt clause.

Resolving a reduced literal requires at least the literals to be present in the reduced clause that were present when the reduction was added to the model assumption. Therefore, resolution cannot act in an arbitrary order, but the literal which was reduced first has to be always chosen next.

However, it is not necessary to resolve all reduced literals in the conflict clauses. Whereas necessary for the initial conflict clause and the justification of a propagation, for the justification of a reduced literal, only those reduced literals are required to be resolved that were false when the reduction was added. To achieve this separation, literals in conflict clauses are marked, and only marked reduced literals need to be resolved. I compute an even smaller sufficient subset by only marking literals which were also not undefined in the reduced clause, i.e., not in $M_1(D)$.

While the number of steps done in the Conflict Phase is exponential in the length of the model assumption, the implementation has a linear run time by using dynamic programming. This is possible because the conflict clause added by explaining a reduced literal only depends on the model assumption and the two involved clauses, but not its context in the conflict clause list. Therefore, it is not necessary to resolve a reduced literal more than once.

3.4 Non-linear Conflict Analysis

Let us return to the example in Figure 2.2, but we now use the RCDCL calculus instead (Figure 3.3). In the first step the clause A reduces $\bar{l}_2$ in B starting unit propagation eventually leading to a conflict in clause D. Normally, this is enough to reach the fail state. However, this is ignored here to show conflict analysis instead.

The first resolution on literal l_2 is the same as it were in CDCL except it is done in two separate steps. Afterward, resolving on l_1 though is interrupted by the reduced conflict literal $\bar{l}_{2B}$, which requires to be resolved beforehand.

$$
\begin{array}{rlll}
 & (\ \varepsilon & ;N; \ \top &)\ \text{start state} \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A} & ;N; \ \top &)\ A \text{ reduces } \bar{l}_2 \text{ in } B \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A}\ l_1^{B} & ;N; \ \top &)\ B \text{ propagates } l_1 \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A}\ l_1^{B}\ l_2^{C} & ;N; \ \top &)\ C \text{ propagates } l_2 \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A}\ l_1^{B}\ l_2^{C} & ;N; \ \bar{l}_{1D} \vee \bar{l}_{2D} &)\ D \text{ is a conflict} \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A}\ l_1^{B} & ;N; \ \bar{l}_{1D} \vee \bar{l}_{2D}^{*};\ \bar{l}_1 \vee l_2 &)\ C \text{ explains } \bar{l}_2 \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A}\ l_1^{B} & ;N; \ \bar{l}_{1D} &)\ \text{Resolution} \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A} & ;N; \ \bar{l}_{1D}^{*};\ l_1 \vee \bar{l}_{2B} &)\ B \text{ explains } \bar{l}_1 \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A} & ;N; \ \bar{l}_{1D}^{*};\ l_1 \vee \bar{l}_{2B}^{*};\ l_1 \vee l_2 &)\ A \text{ explains } \overline{l_{2B}} \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A} & ;N; \ \bar{l}_{1D}^{*};\ l_1 &)\ \text{Resolution} \\
\Rightarrow_{RCDCL} & (\ l_{2B}^{A} & ;N; \ \bot &)\ \text{Resolution} \\
\Rightarrow_{RCDCL} & (\ \varepsilon & ;N; \ \bot &)\ \text{Fail}
\end{array}
$$

Figure 3.3: An RCDCL derivation of the CNF formula $N := A \wedge B \wedge C \wedge D$, where $A := l_1 \vee l_2$, $B := l_1 \vee \bar{l}_2$, $C := \bar{l}_1 \vee l_2$, and $D := \bar{l}_1 \vee \bar{l}_2$.

As its explanation clause A does not add any more reduced conflict literals, the remaining resolutions can then be completed successively to derive the empty clause.

$$
\cfrac{\cfrac{l_1 \vee l_2 \qquad l_1 \vee \bar{l}_2}{l_1} \qquad \cfrac{\bar{l}_1 \vee l_2 \qquad \bar{l}_1 \vee \bar{l}_2}{\bar{l}_1}}{\bot}
$$

Figure 3.4: Derivation of the empty clause in Figure 3.3.

The tree structure of the resolution is shown in Figure 3.4. The interruption by the reduced conflict literal creates a separate branch in the resolution. In CDCL on the other hand, this never occurs within an analysis of a single conflict. A learnt clause can always be derived by a sequence of resolutions, where the intermediate resolvent is resolved with each clause involved in the conflict in the order given by the model assumption.

However, this is not possible when using subsumption resolution. Trying the same as in CDCL leads to an undesired resolvent instead (Figure 3.5). In this case no linear sequence of resolutions is even possible.

Therefore, it is necessary for RCDCL to create these non-linear resolution trees during conflict analysis. I decided to solve this problem using a stack of conflict clauses, as is describe in the rules of RCDCL.

$$\dfrac{l_1 \vee l_2 \qquad \dfrac{l_1 \vee \bar{l}_2 \qquad \dfrac{\bar{l}_1 \vee l_2 \qquad \bar{l}_1 \vee \bar{l}_2}{\bar{l}_1}}{\bar{l}_2}}{l_1 \neq \bot}$$

Figure 3.5: Incorrect Derivation strictly following the model assumption

4 Properties of RCDCL

In the following chapters, many lemmas will be proven by induction over the length of a derivation from the start state to some intermediate state. For simplicity and readability, these proofs will be separately shown for each lemma. However, this will make it seem as though some proofs have circular dependencies. Therefore, it is necessary to consider that in a formal proof all affected lemmas would be proven in parallel, where the circularity is solved by using the strengthened inductive hypothesis.

Furthermore, in the following inductive proofs, cases will be skipped for either of two reasons: Either the rule is not applicable or it does not change any parameters of the proof. In the latter case, it can be immediately closed using the inductive hypothesis.

4.1 Consistency of the Model Assumption

The Consistency of the Model assumption M, defined and proven as invariants in the following three lemmas, consists of two aspects.

The first is a generalization of the CDCL consistency that the Model assumption does not contain conflicting assignments, i.e., a literal being both true and false at once. Reduced literals add complexity to the matter as now a clause specific literal can have conflicting values. A literal can be reduced in a clause and is then considered false in this clause, but can still become true in all other clauses by propagation or decision. Hence, the definition of satisfiability excludes reduced literals from making a clause true.

Secondly, consistency means that the conditions that were holding when a propagation, reduction, or subsumption was added to the model continue to hold until they are again removed by backtracking, and especially when the literals are involved in the analysis of a conflict.

The proofs are mostly trivial, even in the cases which are involved. Only exception is the consistency of the propagation literal added by the Continue Rule, where we anticipate the second main invariant to prove that the learnt clause is indeed productive.

Lemma 4.1.1 (Consistency of M 1)**.** *Let $(M; N; s)$ be a state reachable from $(\epsilon; N'; \top)$.*

1. *If $M := M_1 \bar{l}^C_D M_2$, then l_D is not false under M_1.*
2. *If $M := M_1 l^\top M_2$ or $M := M_1 l^C M_2$, then l is undefined in M_1.*
3. *M^p contains no complementary literals.*

Proof. (1) By induction on the length of the derivation.

Case "Reduction" :
If M_2 is empty, then by construction $M_1(D) = D' \vee l$ and $M_1 \not\models C$. If l_D were false under M_1, then $l \notin M_1(D)$. Therefore, l_D is not false under M_1. Otherwise, by inductive hypothesis.

(2) By induction on the length of the derivation.

Case "Unit Propagation" :
If M_2 is empty, then by construction $M_1(C) = \{l\}$ and $M_1 \not\models C$. Hence, l is not false in C and neither is $l^\top \in M_1$ nor $l^{C'} \in M_1$ as this would satisfy the C. Therefore, l is undefined in M_1. Otherwise, by inductive hypothesis.

Case "Decide" :
If M_2 is empty, then by construction l is undefined in M_1.
Otherwise, by inductive hypothesis.

Case "Continue" :
If M_2 is empty, then by inductive hypothesis on the previous state $(M_1\bar{l}^\top, N\backslash\{C\}, C)$, l is undefined in M_1. Otherwise, by inductive hypothesis.

(3) Assume there are complementary literals l and $\bar{l}$ in M_p.
W.l.o.g. let $M = M_1\, l^\top\, M_2\, \bar{l}^\top\, M_3$. By the result of (2) l is undefined in $M_1 l^\top M_2$. Contradiction. Therefore, M_p contains no complementary literals. □

Lemma 4.1.2 (Consistency of M 2)**.** *Let $(M_1 l^C_D M_2; N; s)$ be a state reachable from $(\epsilon; N'; \top)$. Then, $M_1(C)\backslash\{l\} \subseteq M_1(D)\backslash\{\bar{l}\}$.*

Proof. By induction on the length of the derivation.

Case "Reduction" :
If M_2 is empty, then $M_1(C)\backslash\{l\} \subseteq M_1(D)\backslash\{\bar{l}\}$ holds by construction. Otherwise, by inductive hypothesis. □

Lemma 4.1.3 (Consistency of M 3)**.** *Let* $(Ml^{C \vee l}; N; s)$ *be a state reachable from* $(\epsilon; N'; \top)$*. Then,* $M(C \vee l) = \{l\}$*.*

Proof. By induction on the length of the derivation.

Case "Unit Propagation" :
$M(C \vee l) = \{l\}$ holds by construction.

Case "Continue" :
By assumption, the previous state was $(M\bar{l}^{\top}; N; C \vee l_D)$. By lemma 4.1.1 $\bar{l}^{\top} \notin M$ and $\bar{l}^{C'} \notin M$. Additionally, $\bar{l}^{C'}_{C_p \vee l} \notin M$ by construction. Hence, l is not false under M; therefore, $l \in M(C_p \vee l)$.
Let $l' \in C_p \vee l$ and $l' \neq l$. By lemma 4.2.1 there is an $l'_E \in C \vee l_D$. By invariant 2 (lemma 4.3.3) l'_E is false under $M\bar{l}^{\top}$. Thus, l'_E is false under M because $l' \neq l$. Therefore, $l' \notin M(C_p \vee l)$ and $M(C_p \vee l) = \{l\}$. □

As mentioned at the beginning of the chapter, this is where the circular reasoning appears. In the "Continue" case we use lemma 4.3.3, which itself requires this lemma. However, with the strengthened inductive hypothesis the result of lemma 4.3.3 can be used on any previous states in the derivation, as is the case here.

4.2 Consistency of the Conflict Clauses

The consistency of the list of conflict clauses O is the counterpart to the consistency of M. The following five lemmas are the setup for the two main invariants.

The first lemma proves that the bottom conflict clause, which is the actual conflict clause and at the end becomes the learnt clause, is fully marked as mentioned before. This was already used in the previous lemma and will again become useful to prove the correctness of the fail state.

The second lemma states that any reduced literals in a conflict clause share the same origin. In particular, resolution does not add any other reduced literals due to its restriction.

In the third and fifth lemma consistency again means that two conditions that held when the Explain Reduction Rule was applied continue to hold until the clause is used by the Resolution Rule: First, there is still no reduced literal "left" of the chosen literal, and second, the unmarked literals in a conflict clause contain the counterpart of the chosen literal and the rest are a subset of its preceding clause.

Lastly, the fourth and biggest lemma essentially proves the correctness of the Explain Reduction Rule. It shows that for each marked reduction literal in any conflict clause, its origin clause at the time of the reduction is a subset of the current conflict clause. This tells us that each reduced literal can always be resolved out of the conflict clause by resolution with the same clause that reduced it in its original clause.

Lemma 4.2.1 (Consistency of O 1)**.** *Let* $(M; N; C_0, O)$ *be a state reachable from* $(\epsilon; N'; \top)$*. Then, all literals in* C_0 *are marked.*

Proof. By induction on the length of the derivation.

Case "Conflict" :
By assumption, $C_0 = C_C$ and $O = \varnothing$.
By construction, all literals in C_C are marked.

Case "Resolution" :
By assumption, $C_0, O = O', D' \vee C^m$.
If O' is empty, then $C_0 = D' \vee C^m$. As $D' \subseteq D' \vee \bar{l}^*_D$, all literals in D' are marked by the inductive hypothesis. By definition, C^m only contains marked literals, too. Therefore, all literals in $D' \vee C^m$ are marked.
If O' is not empty, the proof is closed by the inductive hypothesis as C_0 does not change. □

Lemma 4.2.2 (Consistency of O 2)**.** *Let* $(M; N; O)$ *be a state reachable from* $(\epsilon; N'; \top)$*. If for some* $C \in O$ *both* $l_{D_1} \in C$ *and* $l'_{D_2} \in C$ *are reduced under* M*, then* $D_1 = D_2$*.*

Proof. By induction on the length of the derivation.

Case "Conflict" :
By assumption, $O = C_C$. By construction, for all $l_{D_1} \in C_C$ and $l'_{D_2} \in C_C$, $D_1 = C = D_2$.

Case "Explain Propagation" :
By assumption, $O = D' \vee \bar{l}^*_D, C_{C\vee l} \vee l$. By the inductive hypothesis the claim is only left to show for $C_{C\vee l} \vee l$. Let $l'_{D_1} \in C_{C\vee l} \vee l$ and $l''_{D_2} \in C_{C\vee l} \vee l$. By construction, $D_1 = C \vee l = D_2$.

Case "Explain Reduction" :
By assumption, $O = O', D' \vee \bar{l}^*_D, F_{C\vee l} \vee G \vee l$. By the inductive hypothesis the claim is only left to show for $F_{C\vee l} \vee G \vee l$. Let $l'_{D_1} \in F_{C\vee l} \vee G \vee l$ and

$l''_{D_2} \in F_{C \vee l} \vee G \vee l$. By construction, $D_1 = C \vee l = D_2$.

Case “Resolution” :
By assumption, $O = O', D' \vee C^m$. By the inductive hypothesis the claim is only left to show for $D' \vee C^m$. Let $l'_{D_1} \in D' \vee C^m$ and $l''_{D_2} \in D' \vee C^m$ be reduced under M. By assumption, C^m contains no literals reduced under M. Hence, $l'_{D_1}, l''_{D_2} \in D' \subset D' \vee l^*_D$. By the inductive hypothesis $D_1 = D_2$.

Case “Backtrack1-4” :
By assumption, $O = C$. Let $l'_{D_1} \in C$ and $l''_{D_2} \in C$ be reduced under M. By construction, l'_{D_1} and l''_{D_2} are also reduced under $M*$, where $*$ is $l^{C'}_D$, $l^{C'}$, $l^\top$ or $S^{C'}_D$ respectively. By the inductive hypothesis $D_1 = D_2$. □

Lemma 4.2.3 (Consistency of O 3)**.** *Let $(M; N; O_1, D' \vee l^*_D, O_2)$ be a state reachable from $(\epsilon; N'; \top)$. If $l'_E \in D'$ is reduced under a prefix M_1 of M, then l_D is false under M_1 and $D = E$.*

Proof. By induction on the length of the derivation.

Case “Explain Propagation” :
By assumption, $O = D' \vee \bar{l}^*_D, C_{C \vee l} \vee l$ and no literal $l'_E \in D' \vee \bar{l}_D$ is reduced under M. Consequently, no literal is reduced under any subset of M.

Case “Explain Reduction” :
By assumption, $O = O', D' \vee \bar{l}^*_D, F_{C \vee l} \vee G \vee l$ and $M = M_1 l^{C \vee l}_D M_2$. By the inductive hypothesis, the claim is only left to show for D'. Let $l'_E \in D'$ be reduced under the prefix M_3. By construction, l'_E is not reduced under M_1. Hence, $M_1 l^{C \vee l}_D$ has to be a prefix of M_3 as otherwise consistency of M (lemma 4.1.1) were violated. Therefore, l_D is reduced under M_3.

As both $l'_E \in D' \subseteq D' \vee \bar{l}^*_D$ and $l_D \in D' \vee \bar{l}^*_D$ are reduced under M, $E = D$ by lemma 4.2.2. □

Lemma 4.2.4 (Consistency of O 4)**.** *Let $(M; N; O)$ be a state reachable from $(\epsilon; N'; \top)$. If the marked literal l_D in conflict clauses $C \in O$ is reduced under a prefix M_1 of M, then $M_1(D) \subseteq C_p$.*

Proof. By induction on the length of the derivation.

Case “Conflict” :
By assumption, $O = C_C$. Let $l_C \in C_C$ be reduced under M_1. $M_1(C) \subseteq (C_C)_p = C$ holds by definition.

Case "Explain Propagation" :
By assumption, $O = D' \vee \bar{l}^*_D, C_{C\vee l} \vee l$. By the inductive hypothesis the claim is only left to show for $C_{C\vee l} \vee l$. Let $l'_{C\vee l} \in C_{C\vee l} \vee l$ be reduced under M_1. $M_1(C \vee l) \subseteq (C_{C\vee l} \vee l)_p = C \vee l$ holds by definition.

Case "Explain Reduction" :
By assumption, $O = O', D' \vee \bar{l}^*_D, F_{C\vee l} \vee G \vee l$. By the inductive hypothesis the claim is only left to show for $F_{C\vee l} \vee G \vee l$. Let $l'_{C\vee l} \in F_{C\vee l} \vee G \vee l$ be reduced under M_1. $M_1(C \vee l) \subseteq (F_{C\vee l} \vee G \vee l)_p = C \vee l$ holds by definition.

Case "Resolution" :
By assumption, $O = O', D' \vee C^m$ and the previous state was $(M; N; O', D' \vee l^*_D, C)$. By the inductive hypothesis the claim is only left to show for $D' \vee C^m$. Let $l'_E \in D' \vee C^m$ be reduced under M_1. Use case distinction on $D' \vee C^m$.

In the first case $l'_E \in D' \subseteq D' \vee l^*_D$. Thus, by the inductive hypothesis $M_1(E) \subseteq (D' \vee l^*_D)_p = D'_p \vee l$. By lemma 4.2.3 l_D is false under M_1 and $D = E$. Hence, $l \notin M_1(E)$ and $M_1(E) \subseteq D'_p \subseteq (D' \vee C^m)_p$.

In the second case $l'_E \in C^m \subseteq C$. We have as an assumption that no literal in C is reduced under M. This contradicts with l'_E being reduced under $M_1 \subseteq M$.

Case "Backtrack1-4" :
By assumption, $O = C$. Let $l_D \in C$ be reduced under the prefix M_1 of M. $M \subseteq M*$, where $*$ is $l^{C'}_{D'}$, $l^{C'}$, $l^\top$ or $S^{C'}_{D'}$ respectively. Therefore, by the inductive hypothesis $M_1(D) \subseteq C_p$. □

Lemma 4.2.5 (Consistency of O 5)**.** *Let $(M; N; O_1, D' \vee \bar{l}^*_D, C, O_2)$ be a state reachable from $(\epsilon; N'; \top)$. Then, $C^u = C' \vee l$ and $C' \subseteq D'_p$.*

Proof. By induction on the length of the derivation.

Case "Explain Propagation" :
By construction O_1 and O_2 are empty and $C^u = \{l\}$. Therefore, $C' = \emptyset \subseteq D$.

Case "Explain Reduction" :
If O_2 is empty, then by construction the previous state was $(M_1 l^{C\vee l}_D M_2; N; O, D' \vee \bar{l}_D)$ and $(C \vee l)^u = (C \cap M_1(D)) \vee l$. By lemma 4.2.4 $M_1 l^{C\vee l}_D(D) \subseteq D'_p \vee \bar{l}$ and then by definition, $M_1(D) \subseteq D'_p \vee \bar{l}$.
Therefore, $C' = (C \cap M_1(D)) \subseteq M_1(D)\backslash\{\bar{l}\} \subseteq D'_p$.

Otherwise, the claim follows from the inductive hypothesis.

Case “Resolution” :
If O_2 is empty, then by construction $\bar{l}^*_D$ is marked and C^m only contains marked literals. Hence, $(D' \vee \bar{l}^*_D)^u = (D' \vee C^m)^u$. Therefore, the claim follows from the inductive hypothesis.
Otherwise, the claim follows from the inductive hypothesis. □

4.3 Correctness of the Conflict Analysis

The first invariant has a similar counterpart for CDCL: Every conflict clause in O is entailed by the clause set N. This allows adding the resolvent as a new learnt clause without changing the satisfiability of N. The proof simply states that every conflict clause in the list is either an actual member of N or a resolvent of two conflict clauses.

Lemma 4.3.1 (Invariant 1)**.** *Let $(M; N; O)$ be a state reachable from $(\epsilon; N'; \top)$. Then, $N \models C_p$ for all clauses $C \in O$.*

Proof. By induction on the length of the derivation.

Case “Conflict” :
By assumption, $O = C_C$ and $(C_C)_p = C \in N$. Hence, $N \models (C_C)_p$.

Case “Explain Propagation” :
By assumption, $C \vee l \in N$. Hence, $N \models C \vee l$. By inductive hypothesis $N \models (D' \vee \bar{l}_D)_p$. Therefore, $N \models C'_p$ for all $C' \in D' \vee \bar{l}_D, C_{C \vee l} \vee l$.

Case “Explain Reduction” :
By assumption, $O = O', D' \vee \bar{l}^*_D, F_{C \vee l} \vee G \vee l$ and $(F_{C \vee l} \vee G \vee l)_p = C \vee l \in N$. Hence, $N \models (F_{C \vee l} \vee G \vee l)_p$. By inductive hypothesis $N \models C'_p$ for all $C' \in O', D \vee \bar{l}_D$. Therefore, $N \models C'_p$ for all $C' \in O$.

Case “Resolution” :
By inductive hypothesis $N \models C'_p$ for all $C' \in O, D' \vee \bar{l}^*_D, C$. By lemma 4.2.5, $C^u = C' \vee l$ and $C' \subseteq D'_p$. As hence $N \models D'_p \vee \bar{l}$ and $N \models C_p = (C^m \vee C' \vee l)_p$, $N \models D'_p \vee C^m_p = (D' \vee C^m)_p$ follows by resolution. Therefore, $N \models C'_p$ for all $C' \in O, D' \vee C^m$. □

The second invariant shows that every marked literal in a conflict clause is false under M. This invariant was already mentioned several times and even

used once in a preceding lemma, where it was essential in proving correctness of clause learning. Together with lemma 4.2.1 it is also used in proving that the fail state indeed indicates unsatisfiability.

Proving the invariant requires an additional stronger lemma which states that marked literals are not only false under M but are already false under a prefix of M. This prefix corresponds to the state of the model assumption when the reduction of the preceding chosen literal was added.

This lemma also indicates that this relevant prefix of M is getting strictly smaller the closer we get to the top of the conflict clause list. Thereby, we ensure that there are no cyclic dependencies among reductions, and consequently, the length of the list is bounded by the length of M. I will use this property again in the termination proof.

Lemma 4.3.2 (Invariant 2a)**.** *Let $(M; N; O_1, D' \vee l^*_D, C, O_2)$ be a state reachable from $(\epsilon; N'; \top)$. There is a prefix M_1 of M such that l_D is not false under M_1, whereas l'_E is false under M_1 for all $l'_E \in C^m$.*

Proof. By induction on the length of the derivation.

Case "Explain Propagation" :
By assumption $O = D' \vee \bar{l}^*_D, C_{C \vee l} \vee l$, $M = M_1 l^{C \vee l}$ and $\bar{l}_D$ is not reduced under M_1. Assume $\bar{l}_D$ is false under M_1. Then, either $l^\top$, $l^{C'}$ or $l^{C'}_D$ are in M_1. The first two cases contradict consistency of $M_1 l^{C \vee l}$, while the third contradicts the assumption.
Let $l'_{C \vee l} \in C_{C \vee l}$. By lemma 4.1.3, $M_1(C \vee l) = \{l\}$ and consequently, $l' \notin M_1(C \vee l)$. Hence, $l' \in C \vee l$ is false under M_1 and $l'_{C \vee l}$ is false under M_1.

Case "Explain Reduction" :
If O_2 is not empty, this case is proven by the inductive hypothesis.
Otherwise, by assumption $O = O', D' \vee \bar{l}^*_D, F_{C \vee l} \vee G \vee l$ and $M = M_1 l^{C \vee l}_D M_2$.
By consistency of M, $\bar{l}_D$ is not false under M_1.
By lemma 4.1.2, $M_1(C \vee l) \backslash \{l\} \subseteq M_1(D) \backslash \{\bar{l}\}$ and thus, $M_1(C \vee l) \vee \bar{l} \subseteq M_1(D) \vee l$. Hence, $F = C \backslash M_1(D) = (C \vee l) \backslash (M_1(D) \vee l) \subseteq$
$(C \vee l) \backslash (M_1(C \vee l) \vee \bar{l}) = (C \vee l) \backslash M_1(C \vee l)$. By the inductive hypothesis it is only left to show that $l'_{C \vee l}$ is false under M_1 for all $l'_{C \vee l} \in F_{C \vee l} = C^m$.
Let $l'_{C \vee l} \in F_{C \vee l}$. Consequently, $l' \notin M_1(C \vee l)$ and hence, $l' \in C \vee l$ is false under M_1. Therefore, $l'_{C \vee l}$ is false under M_1.

Case "Resolution" :
If O_2 is not empty, this case is proven by the inductive hypothesis.
Otherwise, by assumption $O = O', D'_1 \vee l1^*_{D_1}, D'_2 \vee C^m$ and the previous state is $(M; N; O', D'_1 \vee l1^*_{D_1}, D'_2 \vee l2^*_{D_2}, C)$.

By the inductive hypothesis, there are two prefixes M_1 and M_2 of M such that l_{1D_1} is not false under M_1 and l_{2D_2} is false under M_1, but is not false under M_2. Hence, M_2 is a prefix of M_1 as otherwise consistency of M is violated (lemma 4.1.1).

Let $l_E \in D_2' \vee C^m$. If $l_E \in D_2'$, then by the inductive hypothesis l_E is false under M_1. If $l_E \in C^m$, then by the inductive hypothesis l_E is false under M_2 and under M_1 as well. □

Lemma 4.3.3 (Invariant 2)**.** *Let $(M; N; O)$ be a state reachable from $(\epsilon; N'; \top)$. All marked literals l_D in clauses of O are false under M.*

Proof. By induction on the length of the derivation.

Case "Conflict" :
By assumption, $M(C) = \varnothing$. Hence, l is false under M for all $l \in C$.

Case "Explain Propagation" :
By assumption, $O = D \vee \bar{l}^*_D, C_{C\vee l} \vee l$.
By the inductive hypothesis the claim is only left to show for $C_{C\vee l} \vee l$. Let $l'_{C\vee l} \in C_{C\vee l}$. By lemma 4.3.2 $l'_{C\vee l}$ is false under some $M_1 \subseteq M$.

Case "Explain Reduction" :
By assumption, $O = O', D' \vee \bar{l}^*_D, F_{C\vee l} \vee G \vee l$.
By the inductive hypothesis the claim is only left to show for $F_{C\vee l} \vee G \vee l$. Let $l'_{C\vee l} \in F_{C\vee l}$. By lemma 4.3.2 $l'_{C\vee l}$ is false under some $M_1 \subseteq M$.

Case "Resolution" :
By assumption, $O = O', D' \vee C^m$. By the inductive hypothesis it is only left to show that l_E is false under M for all $l_E \in D' \vee C^m$. Let $l_E \in D' \vee C^m$. Use case distinction on $D' \vee C^m$.
In the first case $l_E \in D' \subseteq D' \vee \bar{l'}^*_D$. By the inductive hypothesis l_E is false.
In the second case $l_E \in C^m \subseteq C$. Again, by the inductive hypothesis l_E is false.

Case "Backtrack1" :
By assumption, $O = C$ and $\bar{l}_D \notin C$. Let $l'_{D'} \in C$. By the inductive hypothesis $l'_{D'}$ is false under $Ml^{C'}_D$. Hence, either $l'_{D'}$ is false under M or $l'_{D'} = \bar{l}_D$. In the first case we are done. In the second we have a contradiction.

Case "Backtrack2-3" :
By assumption, $O = C$. Let $l'_D \in C$. By the inductive hypothesis l'_D is false under $M*$, where $*$ is $l^{C'}$ or $l^\top$ respectively. Hence, either l'_D is false under

M or $l' = \bar{l}$.
In the first case we are done. In the second we have a contradiction as by assumption $\bar{l}_D \notin C$.

Case "Backtrack4" :
By assumption, $O = C$ and $\bar{l}_D \notin C$. Let $l'_{D'} \in C$. By the inductive hypothesis $l'_{D'}$ is false under $MS_D^{C'}$. Hence, $l'_{D'}$ is false under M. □

4.4 Model Construction

In CDCL a full conflict-free model assumption can immediately be used as a model to prove satisfiability. In RCDCL however, M now also contains reduced literals and subsumptions that have to be filtered out first. The intuition is that what is then left, namely M^p, is a model.

Without subsumption this would be rather simple as reduced literals can only make literals be false, but never satisfy a clause by themselves. Therefore, reduced literals in M can be simply ignored.

With subsumptions, however, some clauses might be considered satisfied only due to being subsumed despite having no literal set to true. Therefore, in this section this possibility will be excluded following the simple intuition that for every subsumed clause, there exists a subsuming clause which is itself not subsumed, but satisfied by a literal set to true. This literal is then true in the subsumed clause as well.

The proof consists of three parts:

First, I show that for every subsumed clause there exists a subsuming, but itself not subsumed, clause by defining a termination relation between subsuming and subsumed clauses. Similarly to lemma 4.3.2 for reduced literals, I prove that there are no cycles of clauses subsuming each other.

Second, the monotonicity of M in relation to satisfied clauses is defined. This means that once a clause becomes satisfied it stays satisfied, and in particular, if for a clause C and some prefix M_1 of M a literal in $M_1(C)$ is true, $M_2(C)$ contains this literal for any prefix M_2 of M.

Lastly, using induction on the termination relation combined with the monotonicity result, I will show that the satisfying literal in the subsuming clause is actually one of the literals that both clauses share. Hence, the subsumed clause is also satisfied by a true literal.

Definition 4.4.1 (Subsumption relation $<_M$). *Let M be a model assumption, C and D are clauses. Define $D <_M C$ if $S_C^D \in M$.*

Definition 4.4.2 (Termination relation $<_M^t$)**.** *Let M be a model assumption, M_1 and M_2 are prefixes of M, C and D are clauses.*
Define $(M_1, D) <_M^t (M_2, C)$ if

1. $M_1 = M_2 S_C^D$*, i.e.,* $(M_2 S_C^D, D) <_M^t (M_2, C)$ *or*

2. $M_1 = M_2 M'$*,* $M' \neq \varnothing$*, and* $C = D$ *, i.e.,* $(M_2 M', C) <_M^t (M_2, C)$*.*

Lemma 4.4.1. *If there is a chain $C_1 >_M C_2 >_M \cdots >_M C_n$, then there are prefixes $M_1, \ldots, M_n$ of M with $(M_1, C_1) >_M^t (M_2, C_2) >_M^t \cdots >_M^t (M_n S_{C_{n-1}}^{C_n}, C_n)$.*

Proof. By induction on n.

In the base case $n = 2$ and $C_1 >_M C_2$. By definition, $M = M_1 S_{C_1}^{C_2} M_2$. Therefore, $(M_1, C_1) >_M^t (M_1 S_{C_1}^{C_2}, C_2)$.

If $n > 2$, then by the inductive hypothesis on $n - 1$, there are prefixes $M_1, \ldots, M_{n-1}$ of M with $(M_1, C_1) >_M^t (M_2, C_2) >_M^t \cdots >_M^t (M_{n-1} S_{C_{n-2}}^{C_{n-1}}, C_{n-1})$. As $C_{n-1} >_M C_n$, $M = M_n S_{C_{n-1}}^{C_n} M'$ by definition. Hence, $(M_n, C_{n-1}) >_M^t (M_n S_{C_{n-1}}^{C_n}, C_n)$.
Assume $M_n S_{C_{n-1}}^{C_n}$ is a prefix of $M_{n-1} S_{C_{n-2}}^{C_{n-1}}$.
Then, by definition 3.2.3.2 $M_{n-1} \vDash C_{n-1}$. However, by construction $M_{n-1} \nvDash C_{n-1}$. Contradiction. Thus, $M_{n-1} S_{C_{n-2}}^{C_{n-1}}$ has to be a prefix of $M_n S_{C_{n-1}}^{C_n}$ instead.
If $M_{n-1} S_{C_{n-2}}^{C_{n-1}} = M_n$, then $(M_{n-1} S_{C_{n-2}}^{C_{n-1}}, C_{n-1}) >_M^t (M_n S_{C_{n-1}}^{C_n}, C_n)$.
Otherwise, $(M_{n-1} S_{C_{n-2}}^{C_{n-1}}, C_{n-1}) >_M^t (M_n, C_{n-1}) >_M^t (M_n S_{C_{n-1}}^{C_n}, C_n)$. □

Lemma 4.4.2 (Termination of $<_M$)**.** *The relation $<_M$ is a termination relation.*

Proof. Assume there is an infinitely descending chain in $<_M$. Hence, by lemma 4.4.1 there is also an infinitely descending chain in $<_M^t$. However $<_M^t$ is a terminating relation as each step increases the length of the first component, which is bounded by the length of M. Therefore, $<_M$ is a termination relation. □

Lemma 4.4.3 (Monotonicity)**.** *Let M be a model assumption, C and D are clauses.*

1. *If* $l \notin M_1(C)$*, then* $l \notin M_1 M_2(C)$*.*

2. *If* $l \in M_1 l^\top(C)$ *and* $M_1 l^\top M_2$ *is a prefix of* M*, then* $l \in M_1 l^\top M_2(C)$*.*

3. *If* $l \in M_1 l^D(C)$ *and* $M_1 l^D M_2$ *is a prefix of* M*, then* $l \in M_1 l^D M_2(C)$*.*

4. *If $l \in M_1(C)$, $M_1 \models C$, and $M_1 M_2 l^\top$ or $M_1 M_2 l^D$ is a prefix of M, then $l \in M_1 M_2(C)$.*

1. *Proof.* Let $l \notin M_1(C)$. It follows by definition that $l \notin C$, $\bar{l}^\top \in M_1 \subseteq M_1 M_2$, $\bar{l}^D \in M_1 \subseteq M_1 M_2$ or $\bar{l}_C^D \in M_1 \subseteq M_1 M_2$. Therefore, $l \notin M_1 M_2(C)$. □

2. *Proof.* Assume $l \notin M_1 l^\top M_2(C)$. By definition, $l \notin C$, $\bar{l}^\top \in M_1 l^\top M_2$, $\bar{l}^D \in M_1 l^\top M_2$ or $\bar{l}_C^D \in M_1 l^\top M_2$. $l \notin C$ contradicts with $l \in M_1 l^\top(C)$. $\bar{l}^\top \in M_1 l^\top M_2$ and $\bar{l}^D \in M_1 l^\top M_2$ both contradict with the consistency of M (lemma 4.1.1.2). Lastly, either $\bar{l}_C^D \in M_1 l^\top$ or $\bar{l}_C^D \in M_2$. $\bar{l}_C^D \in M_1 l^\top$ contradicts with $l \in M_1 l^\top(C)$, while $\bar{l}_C^D \in M_2$ contradicts with the consistency of M (lemma 4.1.1.1). □

3. *Proof.* Analog. □

4. *Proof.* By induction on the length of M_2.
   If M_2 is empty, the conclusion is identical to the assumption.
   Otherwise, let $M_2 = l_0 M_2'$. If $l_0 = \bar{l}^\top$ or $l_0 = \bar{l}^{D'}$, then M would be inconsistent (lemma 4.1.1.2). If $l_0 = \bar{l}_C^{D'}$, then by construction $M_1 \not\models C$, which contradicts with $M_1 \models C$. In all remaining cases $l \in M_1 l_0(C)$ and $M_1 l_0 \models C$. Hence, by the inductive hypothesis $l \in M_1 l_0 M_2'(C) = M_1 M_2(C)$. □

Lemma 4.4.4. *Let M be a model assumption, C and D are clauses. If either $M = M_1 l^\top M_2$ or $M = M_1 l^D M_2$, and $l \in M_1(C)$, then $l \in M'(C)$ for every prefix M' of M.*

Proof. Let M' be a prefix of M and w.l.o.g. $M = M_1 l^\top M_2$. Then, either $M' = M_1$, M' is a prefix of M_1 or M_1 is a prefix of M'. For $M' = M_1$ the conclusion is the same as the assumption $l \in M_1(C)$. If M' is a prefix of M_1, assume $l \notin M'(C)$. By lemma 4.4.3.1 $l \notin M_1(C)$ contradicting the assumption. If M_1 is a prefix of M' and $M_1 \neq M'$, then $l \in M_1 l^\top(C)$ and $M' = M_1 l^\top M_3$. Hence, by lemma 4.4.3.2 $l \in M_1 l^\top M_3(C) = M'(C)$. □

Lemma 4.4.5. *Let M be a model assumption, which satisfies the set of clauses N and let C be a clause in N. There exists an $l \in M^p$ such that $l \in M'(C)$ for every prefix M' of M.*

Proof. By induction on $<_M$.
By assumption $M \models C$; hence, either $S_C^D \in M$ or there is an $l \in M_1(C)$, where $M = M_1 l^\top M_2$ or $M = M_1 l^D M_2$.

In the second case let, w.l.o.g., $M = M_1 l^\top M_2$ and $l \in M_1(C)$. Hence, $l \in M^p$ and by lemma 4.4.4 $l \in M'(C)$ for every prefix M' of M.

If $S_C^D \in M$, then by construction $M = M_1 S_C^D M_2$ and $M_1(D) \subseteq M_1(C)$. Because $D <_M C$, by the inductive hypothesis there is an $l \in M^p$ such that $l \in M_1(D) \subseteq M_1(C)$. From $l \in M^p$, it follows by definition that $M = M_3 l^\top M_4$ or $M = M_3 l^{D'} M_4$. W.l.o.g., let $M = M_3 l^\top M_4$. Now, either $l^\top \in M_1$ or $l^\top \in M_2$.
If $l^\top \in M_1$, then also $l \in M_3(C)$ as otherwise there is a contradiction using lemma 4.4.3.1. Consequently, by lemma 4.4.4 $l \in M'(C)$ for every prefix M' of M.
If $l^\top \in M_2$, then $M_1 S_C^D$ is a prefix of $M_3 l^\top$, and additionally, $l \in M_1 S_C^D(C)$ and $M_1 S_C^D \models C$. Thus, $l \in M_3(C)$ by lemma 4.4.3.4, and therefore, by lemma 4.4.4 $l \in M'(C)$ for every prefix M' of M. □

Corollary 4.4.6. *Let M be a model assumption and N a set of clauses. If $M \models N$, then $M^p \models N$.*

Proof. Let $M \models N$ and C be an arbitrary clause in N. By lemma 4.4.5 there exists an $l \in M^p$ such that $l \in M'(C)$ for every prefix M' of M. In particular, $l \in \varepsilon(C) = C$. Therefore, $M^p \models C$. □

4.5 Soundness

In this section I can finally prove the first big result: Soundness.

For the first part I already did most of the proof in the previous section, i.e., for the final states of the calculus where all literals are defined but no clause is conflicting the propositional part of M is a model of the initial set of clauses.

For the second case, where the calculus reaches the fail state, I use another lemma, which tells me that when the Fail Rule was applicable, the conflict phase could have instead continued the conflict phase until the model assumption is fully backtracked. As the bottom conflict clause has to be false under M, this means that for an empty model assumption I derived the empty clause. Hence, as the clause set entails the empty clause, it has to be unsatisfiable.

Lemma 4.5.1 (Consistency of N)**.** *Let $(M; N; s)$ be a state reachable from $(\epsilon; N'; \top)$. Then, $N \models N'$ and $N' \models N$.*

Proof. By induction on the length of the derivation.

Case "Continue" :
By the inductive hypothesis $N \models N'$ and $N' \models N$. As $N \subseteq N \cup \{C_p \vee \bar{l}\}$, $N \cup \{C_p \vee \bar{l}\} \models N \models N'$. By lemma 4.3.1 $N \models C_p \vee \bar{l}$. Hence, $N' \models N \models N \cup \{C_p \vee \bar{l}\}$. Therefore, $N \cup \{C_p \vee \bar{l}\} \models N'$ and $N' \models N \cup \{C_p \vee \bar{l}\}$. □

Lemma 4.5.2 (Fail)**.** *Let $(M; N; O, C)$ be a state reachable from $(\epsilon; N'; \top)$. If no $l^{\top}$ occurs in M, a state $(\epsilon; N; C')$ is reachable.*

Proof. By case analysis.
If there is an $l_D \in C$ with $\bar{l}_D \in M$, then "Explain Reduction" is applicable.
Otherwise, $\bar{l}_D \notin M$ for all $l_D \in C$.
If O is not empty, i.e., $O = O', D'$, then "Resolution" is applicable.
Otherwise, O is empty.
If $M = M'\bar{l}_D^{C'}$, then $l_D \notin C$; hence, "Backtrack1" is applicable.
If $M = M'S_D^{C'}$, then "Backtrack4" is applicable.
If $M = M'\bar{l}^{C'}$ and $l_D \in C$, then "Explain Propagation" is applicable.
If $M = M'\bar{l}^{C'}$ and $l_D \notin C$, then "Backtrack2" is applicable.
If $M = M'\bar{l}^{\top}$, there is a contradiction.
If M is empty, then the current state is $(\epsilon; N; C)$.
Therefore, as backtracking without "Continue" terminates by lemma 4.7.3 and none of these rules change N, the state $(\epsilon; N; C')$ is reachable. □

Theorem 4.5.3 (Soundness)**.** .

1. *Let $(M; N; \top)$ be a final state reachable from $(\epsilon; N'; \top)$. N' is satisfiable, and M^p is a model of N'.*

2. *Let $(\epsilon; N; \bot)$ be a final state reachable from $(\epsilon; N'; \top)$, then N' is unsatisfiable.*

1. *Proof.* Assume literal l is undefined in M, then "Decide" would be applicable. Hence, all literals must be defined in a final state.
   Let C be a clause in N. Then, $M(C) \neq \emptyset$ or $M \models C$ as otherwise "Conflict" would be applicable.
   If $M(C) \neq \emptyset$, there is a literal $l \in C$ with $\bar{l}^{\top} \notin M$, $\bar{l}^{D} \notin M$, and $\bar{l}_C^{D} \notin M$ for any clause D. Therefore, $M = M_1 l^{\top} M_2$ or $M = M_1 l^{D} M_2$ for some clause D as l is defined in M. Hence, by the contra position of lemma 4.4.3.1 $l \in M_1(C)$ and $M \models C$. Therefore, $M \models N$, and by lemma 4.4.6 $M^p \models N$. By lemma 4.1.1 M^p contains no complementary literals. Therefore, M_p is a model of N and N' as $N \models N'$ by lemma 4.5.1.

□

2. *Proof.* If $(\epsilon; N; \bot)$ is reachable, then in the previous step must have been a state $(M; N; C)$ where no $l^\top$ occurs in M.
   By lemma 4.5.2 a state $(\epsilon; N; C')$ is reachable from $(M; N; C)$.
   By lemma 4.3.3 there can be no marked literals in C' as the model assumption is empty. However, by lemma 4.2.1 all literals in C' are marked.
   Hence, the clause C' is the empty clause. By lemma 4.3.1 $N \models C'$; thus, N is unsatisfiable. Using lemma 4.5.1 N' is unsatisfiable as well. □

4.6 Redundancy of Learnt Clauses

Next, I show two properties of learnt clauses that are not actually necessary for correctness and termination. However, they directly correspond to properties of CDCL that are attributed to its efficiency. Hence, it is a reassuring result that they carry over to RCDCL.

The first is redundancy. While every learnt clause is redundant in the sense that it is already entailed by the clause set, it here means that the learnt clause is not subsumed by an existing clause.

The second property simply states that a learnt clause is not a tautology, i.e., does not contain complementary literals.

Theorem 4.6.1 (Redundancy)**.** *Assume "Decide" is only used if "Unit Propagation" and "Conflict" are not applicable.*
Let $(M\bar{l}^\top; N; D)$ be a state reachable from $(\epsilon; N'; \top)$.
If "Continue" is applicable, then there is no $C \in N$ with $C \subseteq D_p$.

Proof. As "Continue" is applicable, $D = D' \vee l'_E$ and $\bar{l}_E \notin M$ for all $l_E \in D' \vee l'_E$. Hence, by lemmas 4.2.1 and 4.3.3, $\overline{l'}^\top \in M$ or $\overline{l'}^E \in M$ for all $l' \in D'_p$ (cf. proof 4.1.3 Case "Continue"). Furthermore, by lemma 4.1.1 l is undefined in M. Now, assume there is a clause $C \in N$ with $C \subseteq D_p$.
If $l \in C$ and l is false in C under M, then C is unsatisfiable under M.
If $l \in C$ and l is not false in C under M, then C is a unit clause under M.
If $l \notin C$, then $C \subseteq D'_p$. Hence, C is unsatisfiable under M.
Thus, in all three cases either "Unit Propagation" or "Conflict" was applicable when $\bar{l}^\top$ was added to the model assumption by "Decide". Contradiction. Therefore, there is no $C \in N$ with $C \subseteq D_p$. □

Theorem 4.6.2 (Tautologies)**.** *Let $(M; N; D)$ be a state reachable from $(\epsilon; N'; \top)$. If "Continue" is applicable, then D_p is not a tautology.*

Proof. As "Continue" is applicable, $\bar{l}_E \notin M$ for all $l_E \in D$. Hence, by lemmas 4.2.1 and 4.3.3, $\overline{l'}^\top \in M$ or $\overline{l'}^{C'} \in M$ for all $l' \in L_p$.

Now, assume L_p is a tautology, i.e., there are literals l and $\bar{l}$ in D_p.
Then, $l \in M_p$ and $\bar{l} \in M_p$, which contradicts lemma 4.1.1.3.
Therefore, D_p is not a tautology. □

4.7 Termination

Lastly, in this section I talk about the termination of the calculus. The proof is build on the intuition that the calculus cannot stay indefinitely in either the propagation phase or the conflict phase, and that it can switch only finitely many times between the two.

For the termination of the propagation phase, I first show that the length of the model assumption is bounded. The given bound of $n + 3^n(n + 1)$, however, should not be taken as a worst case estimation. Then, I argue that every step reduces the distance to this bound.

For the termination of the conflict phase I use the aspect I already mentioned in lemma 4.3.2. Marked conflict literals are considered smaller if they became false earlier, and additionally, a chosen literal is smaller than its normal counterpart. Consequently, conflict clauses are ordered by a multiset ordering on their marked literals, and the whole conflict lists are lexicographically ordered. Then, every step of the conflict phase either shortens the Model assumption or makes the conflict clause list smaller.

Lastly, the two phases do not alternate infinitely. To show this, I split the model assumption into its segments between decisions. With every iteration from one Continue to the next, one of the segments becomes longer, while the ones preceding it stay the same, or the model assumption becomes longer as a whole.

Lemma 4.7.1 (Upper bound for M)**.** *Let N' be a set of non-tautologous clauses, where no clause subsumes another, and n is the number of distinct propositional variables in N'. Let $(M; N; s)$ be a state reachable from $(\epsilon; N'; \top)$. The length of M is always smaller or equal to $n + 3^n(n + 1)$.*

Proof. By lemma 4.1.1 each literal l can be defined only once in M, and for all clauses $C \in N$ and literals $l \in C$, at most l_C^D or $\bar{l}_C^D$ is in M. Furthermore, a clause can be subsumed only once. Hence, the length of M can be at most $n + \Sigma_{C \in N}(\text{length}(C) + 1)$. For n propositional literals the length of a non-tautologous clause is at most n, and there can be at most 3^n distinct non-tautologous clauses. As by the theorems 4.6.1 and 4.6.2 "Continue" does not learn any subsumed or tautologous clauses, the size of N is at most 3^n. Therefore, $\text{length}(M) \leqslant n + \Sigma_{C \in N}(\text{length}(C) + 1) \leqslant n + \Sigma_{C \in N}(n + 1) \leqslant n + 3^n(n + 1)$. □

Definition 4.7.1 (Size-function f)**.** *Let (M, N, s) be a state.*
Let n be the number of distinct propositional variables in N.
Then, $f(M) = n + 3^n(n+1) - length(M)$.

Definition 4.7.2 ($>_S$)**.** *Let (M, N, s) and (M', N', s') be two states. Consider a decomposition $M_0 + l_1^\top + M_1 + ... + l_k^\top + M_k$ of M (accordingly for M'). Then, $(M, N, s) >_S (M', N', s')$ if*

1. *$f(M_0) = f(M'_0), ..., f(M_{i-1}) = f(M'_{i-1}), f(M_i) > f(M'_i)$ for some $i \leqslant k, k_0$ or*
2. *$f(M_j) = f(M'_j)$ for all $0 \leqslant j \leqslant k$ and $f(M) > f(M')$ or*
3. *$M = M'$, $s = \top$ and $s' \neq \top$.*

Definition 4.7.3 ($>_O$)**.** *Let (M, N, O) and (M, N, O') be two states.*

1. *Let l_D, l'_C be two conflict literals. Define $l_D >_L l^*_D$ and $l^*_D >_L l'_C$ if $M = M_1M_2$ with l'_C false in M_1, while l_D is not false in M_1.*
2. *Let C and D be two conflict clauses. Define $C >_{Cl} D$ as the multiset ordering of the transitive closure of $>_L$.*
3. *Define $O >_O O'$ as the lexicographic ordering on $>_{Cl}$.*

Note: $l_D >^*_L l'_C$ compares the left-most positions in M, where each literal becomes false under M. Hence, a descending chain is linearly bounded by the length of M. Therefore, $>^*_L$ is a well-founded ordering.

Definition 4.7.4 ($>_B$)**.** *Let (M, N, O) and (M', N', O') be two states.*
Define $(M, N, O) >_B (M', N', O')$ if

1. *$length(M) > length(M')$ or*
2. *$M = M'$ and $O >_O O'$.*

Lemma 4.7.2 (Termination of Propagation)**.** *Let $(M; N; \top)$ be a state reachable from $(\epsilon; N'; \top)$. A derivation starting in $(M; N; \top)$ either terminates without using "Conflict" or reaches a state $(M'; N; C)$ after an application of "Conflict", where $(M; N; \top) >_S (M'; N; C)$.*

Proof. Let $M_0 + l_1^\top + M_1 + ... + l_k^\top + M_k$ be a decomposition of M.
By case analysis every rule-application decreases $>_S$ and
reaches a new state $(M'; N; \top)$ unless the rule was "Conflict".

Case "Unit Propagation" :
By assumption $M' = Ml^C = M_0 + l_1^\top + M_1 + ... + l_k^\top + M_k l^C$. Hence, $f(M_0) = f(M'_0), ..., f(M_{k-1}) = f(M'_{k-1})$ and $f(M_k) > f(M_k l^C) = f(M'_k)$. Therefore, $(M; N; \top) >_S (Ml^C; N; \top)$.

Case "Reduction" :
By assumption $M' = Ml_D^C = M_0 + l_1^\top + M_1 + ... + l_k^\top + M_k l_D^C$. Hence, $f(M_0) = f(M'_0), ..., f(M_{k-1}) = f(M'_{k-1})$ and $f(M_k) > f(M_k l_D^C) = f(M'_k)$. Therefore, $(M; N; \top) >_S (Ml_D^C; N; \top)$.

Case "Subsumption" :
By assumption $M' = MS_D^C = M_0 + l_1^\top + M_1 + ... + l_k^\top + M_k S_D^C$. Hence, $f(M_0) = f(M'_0), ..., f(M_{k-1}) = f(M'_{k-1})$ and $f(M_k) > f(M_k S_D^C) = f(M'_k)$. Therefore, $(M; N; \top) >_S (MS_D^C; N; \top)$.

Case "Decide" :
By assumption $M' = Ml^\top = M_0 + l_1^\top + M_1 + ... + l_k^\top + M_k l^\top$. Hence, $f(M_j) = f(M'_j)$ for all $0 \leqslant j \leqslant k$ and $f(M) > f(Ml^\top) = f(M')$. Therefore, $(M; N; \top) >_S (Ml^\top; N; \top)$.

Case "Conflict" :
By assumption $M = M'$, $s = \top$ and $s' = C_C$.
Therefore, $(M; N; \top) >_S (M; N; C_C)$. □

Lemma 4.7.3 (Termination of Backtracking). *Let $(M; N; O_1)$ be a state reachable from $(\epsilon; N'; \top)$. A derivation starting in $(M; N; O_1)$ either terminates without using "Continue" or reaches a state $(M_0\, l^D; N \cup \{D\}; \top)$ after an application of "Continue", where $M = M_0\, M''$.*

Proof. By case analysis every rule-application decreases $>_B$ and reaches a new state $(M'; N; O_2)$ with $M = M'M''$ unless the rule was "Continue" or "Fail".

Case "Explain Propagation" :
By assumption $M = M'$, $O_1 = D' \vee \bar{l}_D$, and $O_2 = D' \vee \bar{l}_D^*, C_{C \vee l} \vee l$.
As $\bar{l}_D >_L \bar{l}_D^*$, $D' \vee \bar{l}_D >_{Cl} D' \vee \bar{l}_D^*$. Hence, $O_1 >_O O_2$, and therefore, $(M; N; O_1) >_B (M'; N; O_2)$ with $M = M'$.

Case "Explain Reduction" :
By assumption $M = M'$, $O_1 = O, D' \vee \bar{l}_D$, and $O_2 = O, D' \vee \bar{l}_D^*, F_{C \vee l} \vee G \vee l$.
As $\bar{l}_D >_L \bar{l}_D^*$, we have $D' \vee \bar{l}_D >_{Cl} D' \vee \bar{l}_D^*$. Hence, $O_1 >_O O_2$, and

therefore, $(M; N; O_1) >_B (M'; N; O_2)$ with $M = M'$.

Case "Resolution" :
By assumption $M = M'$, $O_1 = O, D' \vee \bar{l}^*_D, C$, and $O_2 = O, D' \vee C^m$.
Let $l'_E \in C^m \subseteq C$. By lemma 4.3.2 there is a prefix M_1 of M such that $\bar{l}_D$ is not false under M_1 and l'_E is false under M_1. Hence, $\bar{l}^*_D >_L l'_E$, $D' \vee \bar{l}^*_D >_{Cl} D' \vee C^m$, and $O_1 >_O O_2$.
Therefore, $(M; N; O_1) >_B (M'; N; O_2)$ with $M = M'$.

Case "Backtrack1-4" :
By assumption $M = M'*$. Where $*$ is either $l^{C'}_D$, $l^{C'}$, $l^\top$ or $S^{C'}_D$ respectively.
Therefore, $(M'*; N; O_1) >_B (M'; N; O_2)$.

Case "Continue" :
By assumption $M = M_0\, l^\top$ and $M' = M_0\, l^{C_p \vee l}$.
Therefore, $(M_0\, l^{C_p \vee l}; N \cup \{C_p \vee l\}; \top)$ is a state after an application of $Continue$.

Case "Fail" :
The new state $(\epsilon; N; \bot)$ is a final state. □

Theorem 4.7.4 (Termination)**.** *Let* $(M; N; \top)$ *be a state reachable from* $(\epsilon; N'; \top)$*. Every derivation starting from* $(M'; N'; \top)$ *terminates.*

Proof. I show that every derivation starting in $(M; N; \top)$ either terminates or reaches a state $(M'; N'; \top)$, where $(M; N; \top) >_S (M'; N'; \top)$.

(1) By lemma 4.7.2 a derivation starting in $(M; N; \top)$ either terminates without using "Conflict" or reaches a state $(M''; N; C)$ after an application of "Conflict", where $(M; N; \top) >_S (M''; N; C)$.

(2) By lemma 4.7.3 a derivation starting in $(M''; N; C)$ either terminates without using "Continue" or reaches a state $(M'''l^D; N \cup \{D\}; \top)$ after an application of "Continue", where $M'' = M'''M''''$.
Let $M_0 + l_1^\top + M_1 + ... + l_k^\top + M_k$ be a decomposition of M''.
By construction of "Continue", $M''' = M_0 + l_1^\top + M_1 + ... + l_{k'}^\top + M_{k'}$ for some $k' < k$. Hence, $f(M_0) = f(M'_0), ..., f(M_{k'-1}) = f(M'_{k'-1})$ and $f(M_{k'}) > f(M_{k'} l^{D \vee l}) = f(M'_{k'})$. Thus, $(M''; N; C) >_S (M'''l^D; N \cup \{D\}; \top)$.

(3) By transitivity $(M''l^D; N \cup \{D\}; \top)$ is reachable from $(\epsilon; N'; \top)$ and $(M; N; \top) >_S (M'''l^D; N \cup \{D\}; \top)$. □

5 Implementation

The goal of the implementation is to test the effect of applying subsumption resolution during the search. Therefore, to better compare the results of RCDCL and CDCL, I avoid using heuristics. The implementation itself is based on a basic version of SPASS-SATT [11].

This chapter focuses on two aspects: Finding subsumption resolutions and the linear implementation of conflict resolution.

5.1 Data Structures

The important data structure that connects both phases is the stack that simulates the model assumption. However, instead of just one stack there are three:

RStack is the reduction stack and contains all decision, propagation, and reduced literals as tuples (*literal*, *justification*, *clause*, *mark*). The *justification* is the clause that either propagated the literal or reduced it in the reduced *clause*. For decisions both *justification* and *clause* are null, whereas only *clause* is for propagation literals. The *mark* is used in the conflict analysis.

SStack is the subsumption stack and consists only of the subsumed clauses. As subsumption is never responsible for a conflict, they are stored separately, such that conflict analysis can ignore them.

PStack is the propagation stack and the same as in a standard implementation of CDCL, i.e., it only contains decisions and propagations.

Next are clauses and the ways they are accessed. Aside from the 2-watched literal data structure, there is an *AllClauses* array and a *Litset* data structure. *AllClauses* maps each clause number to its corresponding clause, while *LitSets* are occurrence sets for each literal. The implementation of *LitSets* is especially adapted for fast intersections while inserting and iterating over elements is still possible. Its implementation is discussed in more detail in Section 5.2.5.

5.2 Propagation Phase

In the previous chapter I have proven the use of subsumption resolutions correct, but left out how to actually find them. For unit propagation the 2-watched literal data structure has been established since Chaff [10]. For subsumption resolution the challenge is to find a data structure which allows a comparably efficient method. However, as subsumption resolution involves comparing pairs of clauses, the process is inherently quadratic.

5.2.1 Propagation

Algorithm 1: Clause Propagation(**int** Index, **int** PIndex)

```
Input: Stack and PStack indices where propagation starts.
Output: NULL if no conflict was detected; otherwise the Conflict.
Conflict = NULL;
while NoConflict and Index < StackSize do
    while NoConflict and PIndex < PStackSize do
        Conflict = UnitPropagation(PStack(PIndex));
        PIndex++;
    if Conflict then return Conflict;
    ;
    switch Stack(Index) do
        case l^⊤ or l^C
            for each Clause in LitSet(−l) do
                Conflict = BackwardSubRes(Clause) ;
        case l_D^C
            Conflict = BackwardSubRes(D) ;
    Index++;
return Conflict;
```

The main loop of propagation is described in Algorithm 1.

The first thing to note is that *UnitPropagation* still uses the 2-watched literal lists. As unit propagation is a special case of subsumption resolution, it will always be more efficient. Furthermore, the effect of propagating a unit is many times larger than that of a single reduction. Therefore, searching for reductions is delayed as much as possible until the inner while loop completes exhaustive unit propagation by iterating over the propagation stack. When

unit propagation is done and no conflict occurred, the outer loop continues its iteration.

Next, to actually look for reductions, I use a method inspired by backward subsumption, namely, pick a clause and check whether it subsumes or reduces another in the remaining set. Here, two assumptions are used. First, the initial set of clauses is completely reduced, i.e., there are no reductions or subsumptions possible. Second, no clauses are added to the problem set. While the first assumption is generally assured by preprocessing, the second is continuously broken with each learned clause. Section 5.2.6 explains how this is counteracted. Under these assumptions it holds that if a clause does not reduce any other clause, this property lasts as long as the clause itself does not change. So, only if the clause shrinks, it could again reduce - or subsume - another clause.

As the stack tells us which clauses have changed and when, I therefore also know which clauses became viable for a backward subsumption resolution check. In the following those are referred to as candidates. In case of decisions or propagations, all clauses with the negated literal and in case of a reduced literal, only the affected clause become candidates.

5.2.2 Backward Subsumption Resolution

Algorithm 2: Clause BackwardSubRes(**Clause** Candidate)

```
Output: Candidate if it is false; otherwise NULL.
if IsSubsumed(Candidate) then return NULL;
;
Seen = ∅;
for each literal L in Candidate do
    switch TruthValue(Candidate,L) do
        case True  return NULL;
        ;
        case False  continue ;
        ;
        case Undefinded
            Seen = Seen ∪{L} ;
if S == ∅ then return Candidate;
for each Partner in FindPartners(Seen, Candidate) do
    SubRes(Candidate, Partner, Seen);
return NULL
```

Algorithm 3: *SubRes*(**Clause** Candidate, **Clause** Partner, **Lits** Seen)

```
if Partner == Candidate or IsSubsumed(Partner) then return;
Literal Reduce = NULL;
Hit = 0;
for each literal L in Partner do
    switch TruthValue(L) do
        case True return;
        ;
        case False continue ;
        ;
        case Undefinded
            if L ∈ Seen then Hit++;
            ;
            if −L ∈ Seen then Reduce = L;
if Reduce ≠ NULL and Hit == |Seen| − 1 then
    Candidate reduces Reduce in Partner
if Hit == |Seen| then
    Candidate subsumes Partner
```

Next, the Algorithms 2 and 3 show how reductions are found.

First, the candidate could already be subsumed. In that case, even if it does reduce another clause, its subsuming clause would either reduce or subsume that clause as well. Hence, it is ignored. The same holds if the candidate is satisfied by a true literal. In most cases due to the two watched literals, a satisfied candidate will be detected early by a true literal in the first or second position. Then, the candidate's undefined literals are added to the *Seen* set, which will later enable a constant time element check.

Next, the clauses to check the candidate against, called *Partners*, are chosen. The naive way - short of taking all clauses - is to pick an atom from *Seen* and take its occurrence set. The more sophisticated function *FindPartners* is described in the next chapter.

Lastly, it performs for each *Partner* the actual check: After again excluding subsumed or satisfied clauses and preventing the *Candidate* from subsuming itself, It simply counts the number of undefined literals that match with *Seen* and notes if a literal appears negated. The set returned by *FindPartners* even prevents the case where more than one literal is negated. After this is done, having counted the same number as the size of *Seen* means every undefined literal in *Candidate* appears in *Partner*, hence *Candidate* subsumes *Partner*. Counting one less but having found a negated literal respectively means that

the *Candidate* reduces this literal in *Partner*.

In the special case that a watched literal is reduced, the same procedure as in unit propagation starts. A still undefined literal replaces the watched literal and the watched lists are updated. In case the clause is now unit, the last undefined literal is also propagated.

5.2.3 Determining a Literal's Truth value

As the truth value of a reduced literal is not global but can differ per clause, reduced literals need to be recognizable. Fortunately, I can achieve this indirectly.

Whenever a literal is reduced, I swap it with the currently last position in the clause and reduce the size value of the clause by one. Iterating over the literals then never encounters reduced literals, which are thereby effectively removed from the clause. When I later backtrack the reduced literal and pop it from the stack, I simply increase the clause's size again to reinclude the literal into the clause.

However, conflict analysis still needs the reduced literals. Hence, I save both sizes, current and actual size, individually. Then, I identify a literal outside the current bound as a reduced literal.

A similar trick is possible for subsumption, where I use a simple bit flag in the clause. The flag is set when the subsumption is found and unset when it is popped from the subsumption stack.

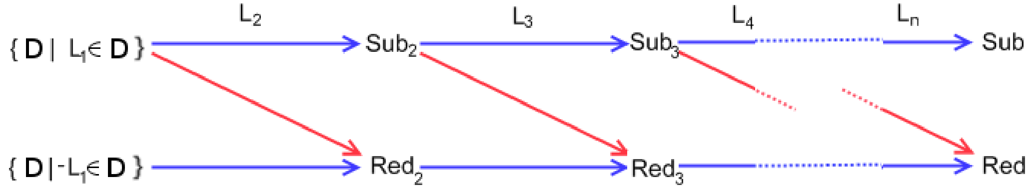
5.2.4 Partner Selection for Subsumption Resolution

Algorithm 4: CSet *FindPartners*(**Lits** *Seen*, **Clause** *Candidate*)

Output: A Set of Clauses subsumed or reduced by *Seen*.
$Sub = AllClauses() \setminus \{Candidate\}$;
$Red = \emptyset$;
for each literal L **in** *Seen* **do**
  $Red = (Sub \cap LitSet(-L)) \cup (Red \cap LitSet(L))$;
  $Sub = Sub \cap LitSet(L)$;
  if $Sub == \emptyset$ **and** $Red == \emptyset$ **then break**;
  ;
return $Sub \cup Red$

The function I skipped in the last section *FindPartners* is given in Algorithm 4. The essential idea is that for a given set *Seen*, I only want those

Figure 5.1: Graphical representation of *FindPartners*



The *Candidate* clause's undefined literals are L_1 to L_n. Blue arrows represent intersections with the occurrence set of the literal on top, whereas red arrows represent intersections with the occurrence set of the negated literal. Note that any path has at most one red arrow. Hence, every clause in the end in *Sub* and *Red* contains at most one atom negated compared to the candidate.

clauses which contain the same atoms with at most one negated. The method is based on the backward subsumption detection by Zhang [14].

The algorithm starts with two sets: *Sub*, the set of all clauses, and *Red*, the empty set. Then, it iterates over the literals in *Seen*, and for each it intersects the sets of clauses containing this literal or its negation with *Sub* and *Red* as specified by Algorithm 4 and Figure 5.1.

After the first iteration, *Sub* and *Red* are the first literal's positive and negative occurrence sets. As this is always the same, the actual implementation starts there instead. After the next iteration, *Sub* has all clauses containing the first and second literal, while *Red* has the ones containing the first or second literal and the other one negated. At the end *Sub* contains all clauses with the same literals as *Seen* and *Red* has the clauses containing exactly one literal negated. As the *Candidate* is guaranteed to stay in *Sub* until the end but will not be used anyway, it is removed at the start.

The big advantage of this approach is that with each iteration *Sub* and *Red* shrink quickly and often become empty long before the last iteration.

Furthermore, the implementation uses several hacks to improve performance. The simplest is to sort *Seen* by the size of the corresponding *LitSets*. The next is called *WordLevelParallelism* [4]. By encoding clauses into bitmaps of machine word size, a single AND instruction intersects an entire range of clauses.

In the case *Sub* and *Red* become empty early, any literal not yet used is irrelevant to the outcome. Again ignoring learnt clauses, as long as none of the used literals change in *Candidate*, *FindPartners* will return the empty set. Hence, I mark the unused literals and ignore the clause as a candidate if one of them becomes false. As the model assumption is ignored, the marks

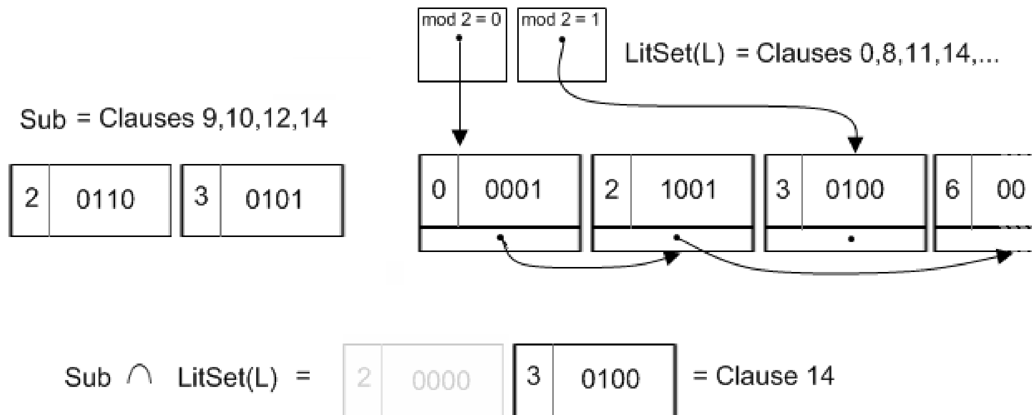


Figure 5.2: Example of the LitSet data structure and its use.
On the left is one of the temporary lists *Sub* and on the right the occurrence set of a literal L. The word size w is 4 and the hash table has length 2. For example, the block $(2, 0110)$ encodes clauses $2*4+1 = 9$ and $2*4+2 = 10$. The result of intersecting the two is shown on the bottom. The intersection can be computed either linearly or looking up the blocks 2 and 3 in *LitSet(L)* using the hash mod 2 and following the pointers. As the intersection of the 2-blocks in both sets – $0110\&1001 = 0000$ – is empty, the block is discarded in the result.

still apply even after backtracking.

After the second or respectively first real iteration, *Sub* and *Red* will already be considerably shorter than the sets they are intersected with. Using hashing, the intersections can be computed in linear time of the sizes of *Sub* and *Red* instead of their combined sizes. Lastly, the three intersections could also be computed concurrently, which is not currently implemented.

5.2.5 The *LitSet* Data Structure

Profiling shows that the *FindPartners* function takes up the largest part of the execution time. Hence, it is very important that intersections are as fast as possible. Hereby, two cases occur. The first is an intersection between sets of balanced sizes, which generally occurs during the first iteration in *FindPartners*. In this case having two sorted arrays and iterating over them simultaneously is considered most efficient. Furthermore using the mentioned *WordLevelParallelism* results in a linear speedup depending on the word size.

With each iteration the intermediate results *Sub* and *Red* shrink, while the *Litsets* they are intersected with increase in size. After the first iteration their sizes are already unbalanced. In that case using a hash set for the *Litsets*

and a simple array for *Sub* and *Red* is faster because using an element check for each element in *Sub* and *Red* ignores the size of the *LitSet*.

I want to take advantage of both cases, while avoiding two separate data structures, which would double the space requirement. Therefore, I combine both into one by using a Chaining Hash set where the entries are stored in a sorted array. This is implemented with three arrays.

The first is the array *Sort*, which contains pairs of block numbers b and bit vectors v and is sorted by b. A clause with number c is contained in a *LitSet* if *Sort* has a pair $(c \div w, v)$ with the $(c \mod w)$-th bit is set in v. Furthermore, in each v at least one bit is set. Thus, if on average more than two bits are set in v we save space and in the worst case we use at most twice as much compared to storing pointers. *Sort* is used for the linear intersection, which is similar to the merge from Merge sort.

The remaining arrays are *Hash* and *Link*, which contain indices of entries in *Sort*. *Hash* has length n and the index stored at position k in *Hash* is the first entry (b, v) in *Sort* such that $b \mod n = k$. *Link* has the same length as *Sort* and it stores at position i the index j if j is the next larger index after i in *Sort* such that $b_i \mod n = b_j \mod n$. In both arrays the default value is -1. To look up an entry $(b, _)$ in *Sort* using hashing get the $(b \mod n)$-th entry from *Hash* and then follow the entries of *Link* until either the entry in *Sort* is found or a -1 index is encountered. In that case there is no entry with block number b in *Sort*.

5.2.6 Propagation after Restart

Let me now revisit propagation. In the beginning I mentioned that I use backward checks as the core aspect of the search for reductions. However, this has one major flaw. The assumption that a clause does not reduce another as long as itself does not change does not take learnt clauses into account. While a learnt clause is guaranteed to not be subsumed, it could still be reduced and reduce or subsume existing clauses. Therefore, with every clause added, this approach becomes less reliable.

The implemented solution consists of two steps. The first step is to use a forward check when a learnt clause is created to ensure that it is at least not reducible by any clause under the empty model assumption. The second step is to regularly restart, where the first propagation after a restart is modified as described in Algorithm 5.

The first difference is that while again prioritizing unit propagation, I first use *BackwardSubRes* on all new learnt clauses since the last restart. As they are now guaranteed to not be reducible by any of the old clauses, this will

```
Algorithm 5: Clause PropagationRestart()
Output: NULL if no conflict was detected; otherwise the Conflict.
PIndex = PStackSizeAtRestart();
Conflict = NULL;
for each Clause in NewClauses() do
    while ¬Conflict and PIndex < PStackSize do
        Conflict = UnitPropagation(PStack(PIndex));
        PIndex++;
    if Conflict then
        return Conflict
    else
        Conflict = BackwardSubRes(Clause) ;
Conflict =Propagation(StackSizeAtRestart(),PIndex) ;
BuildLitSets();
return Conflict;
```

in the end lead to exhaustive reduction and subsumption. This restores the assumption on the clause set mentioned at the beginning of the chapter.

```
Algorithm 6: BuildLitSets()
ClearAllLitSets();
Index = 0;
for each Clause in AllClauses() do
    if IsSubsumed(Clause) or IsSatisfied(Clause) then continue ;
    ;
    for each literal L in Clause do
        switch TruthValue(Candidate,L) do
            case False break ;
            ;
            case Undefined
                AddToLitSet(L,Index) ;
    SetElement(AllClauses(),Index,Clause);
    Index++;
ReduceSize(AllClauses(),Index);
```

As any propagation, reduction, and subsumption found this way is prior to the first decision, they are never backtracked. Hence, the second change

is that before continuing I reset the data structures and build them again (cf. Algorithm 6). Satisfied clauses are removed, the remaining clauses are renumbered and only clauses with an undefined literals are added to the respective *LitSets*.

Aside from the obvious advantages of a smaller problem size, this operation especially benefits the expensive function *FindPartners*. First, the reduced size of the *LitSets* directly speeds up the intersections. Secondly, the reduction of the clause number range due to renumbering improves the effect of the *WordLevelParallelism*. Lastly, as *FindPartners* ignores the model assumption, it returns partner clauses that have already been reduced or subsumed every time the responsible clause is a candidate. This means that those clauses need to be repeatedly checked and prevents *FindPartners* from stopping early. Rebuilding the *LitSets* alleviates this at least in respect to reductions and subsumptions found before the first decision.

On the other hand, there is the disadvantage of additional time consumption. Whereas in the standard implementation the benefit is too small to warrant the work, with reduction the combined benefits for *FindPartners* far outweighs its cost. Furthermore, as the amounts of deleted and remaining clauses is relatively balanced and the *LitSets* are sorted by clause number, removing deleted entries instead of rebuilding the data structures from scratch is actually more expensive aside from being harder to implement.

5.3 Conflict Phase

The second focus of this chapter is the Conflict phase. As has been seen in the RCDCL and Proof Chapters, reduction adds a layer of complexity to analyzing a conflict, for its implementation as well as its run time. Hence, in this section I will describe how it is implemented in almost linear run time complexity in the size of the model assumption.

5.3.1 Conflict Analysis

The most important thing to note is that contrary to the description of the calculus, there is no list of conflict clauses, not even a single conflict clause. Instead, everything is encoded by the marks in the entries of the *RStack*. The marks express three states an entry can be in. The first is *NotMarked*, which is the default value. The next is *Marked*. Marked entries correspond to marked literals in the conflict clause list. Lastly there are *WasMarked* entries. Those encode conflict literals that were marked once but have since been resolved away.

Algorithm 7: Clause *Resolution(***Clause** *Conflict)*

```
Input: A false Clause Conflict.
Output: A Resolvent without reduced Literals.
Resolvent = ∅;
PIndex = PStackTop();
RIndex = 0;
for each literal l in Conflict do
    Mark( l_Conflict);
while PIndex ⩾ 0 do
    while RIndex ⩽ RStackTop() do
        if RStack(RIndex)== l_D^{C∨l} and Marked(l_D^{C∨l}) then
            for each literal L in C\M_RIndex(D) do
                if ¬ WasMarked(L_{C∨l}) then
                    Mark(L_{C∨l}) ;
                    RIndex = −1;
            UnMark( l_D^{C∨l});
        RIndex++;
    switch  PStack(Index) do
        case l^⊤
            if Marked(l^⊤) then break ;
            ;
        case l^{C∨l}
            for each literal L in C do
                if ¬ WasMarked(L_{C∨l}) then
                    Mark(L_{C∨l});
                    RIndex = 0;
            UnMark(l^{C∨l});
    PIndex−− ;
while PIndex ⩾ 0 do
    if Marked(PStack(PIndex)) then
        Resolvent = Resolvent ∪ {PStack(PIndex)}
    PIndex−− ;
return ForwardReduction(Resolvent);
```

For unmarked conflict literals, such strong invariants hold that they do not have to be modeled at all.

The *Resolution* is described in Algorithm 7. The first step is to encode the conflict clause. For every literal the corresponding entry on the *RStack* is marked. Stack offsets saved in the clauses for reduced literals and in the assignment for decision and propagation literals enable constant access to these entries.

Then analogously to *Propagation* but in reversed order, the inner loop first resolves all reduced literals by iterating over the *RStack* bottom up, while the outer loop resolves propagation literals iterating top down.

In each case I respectively mark the entries corresponding to the marked conflict literals as described in the calculus and unmark the resolved literal. However, if an entry was already marked, it is not marked again. This way I never resolve a reduced literal twice. While the loops themselves have quadratic run time, actual computation only occurs in linearly many cases.

Lastly, the learnt clause is extracted by collecting all marked literals from the stack and is added to the clause set after calling *ForwardReduction*.

The implementation also uses 1UIP [12], which is not reflected here in the pseudo code. However, if the number of top level conflict literals is checked after the inner loop finishes, reduced literals on the top level do not have to be counted.

5.3.2 Forward Reduction of Learned Clauses

Algorithm 8: Clause *ForwardReduction*(**Clause** *Resolvent*)

```
Output: An irreducible Resolvent.
for each literal L in Resolvent do
    for each Clause in PropSet(−L) do
        if Reduces(Clause, Resolvent, L) then
            Resolvent = Resolvent\{L};
            break ;
return Resolvent
```

As mentioned in Section 5.2.6, an assumption of propagation after a restart is that new clauses are neither reduced nor subsumed by any existing clause. Whereas subsumption is excluded by Lemma 4.6.1, reductions are still possible. Hence, I use *ForwardReduction* (Algorithm 8) on any new learned clause.

Algorithm 9: Bool *Reduces*(**Clause** *Candidate*, *Clause*, **Literal** *Lit*)

Input: $-Lit \in Candidate$ and $Lit \in Clause$.
Output: Whether *Candidate* reduces *Lit* in *Clause*.

```
for each literal L in Candidate do
    if L ≠ −Lit and L ∉ Clause then
        return False
return True
```

ForwardReduction iterates over each literal and checks whether a clause containing the negated literal reduces it. Note the fact that any clause reducing a learnt clause has exactly one true literal, while the remaining literals are false. By the invariant of the 2-watched literal data structure in such a clause this true literal is watched. Hence, it actually only needs to iterate over the set of clauses where the literal to be reduced is watched.

Additionally, as the literals of the *Resolvent* are ordered by their decision level, it can skip the first literal. Assuming the first literal, which has the highest decision level in the clause, were reducible the search would have missed either a conflict or a unit propagation when the literal had been decided.

Furthermore, reducing the second and following literals can result in a farther back-jump than was possible without *ForwardReduction*.

6 Results

6.1 The Effect of Subsumption on Satisfied Clauses

As mentioned, the *Subsumption* Rule is not essential to the calculus. One can even see in the rules and proofs that subsumptions are not involved with conflicts and therefore, do not influence the search depth.

On the other hand, in the implementation subsumptions save time as they exclude clauses from becoming candidates. At the same time finding them creates little overhead. Ignoring subsumptions would only remove a single intersection from *FindPartners* in some cases and their respective checks. The only apparent drawback is the added complexity in the soundness proof.

However, without subsumptions I could use a simpler definition 3.2.3 for satisfied clauses; namely the same definition as in CDCL: A clause is satisfied whenever any literal is set to true; even one that had been reduced already.

Intuitively, this can be explained as reducing a literal does not actually set it to false in the respective clause, but it is rather considered irrelevant. More formally, if a reduced literal is set to true, it becomes false in the reducing clause, which then subsumes the reduced clause. Even if further reductions prevent an immediate subsumption, one could derive a resolvent from the involved clauses that does.

The last question left is why this definition cannot be used in combination with subsumption. The answer is that together they can create cycles between subsuming clauses. For example, assume the clauses $L_1 \vee L_2 \vee L_3$ and $\overline{L}_1 \vee L_2 \vee L_3$, where the first reduces $\overline{L}_1$ in the second and then the second subsumes the first. Now assume we decide $\overline{L}_1$, which would satisfy the second clause. It seems both are satisfied, although the model $\overline{L}_1$ by itself only satisfies the second. The decision actually invalidated the subsumption.

Therefore, to prevent this I restrict the definition to literals that are not reduced.

6.2 Pure Literal Elimination

Another theoretical point are pure literals. At an earlier stage, a more naive implementation of propagation detected some pure literals with only a constant overhead. This was, however, dropped in favor of the *FindPartners* function.

One particular difference is that without pure literals I can use subsumed clauses as the conflict clause. While being subsumed guarantees that there is another conflicting clause which is not subsumed, either one can be used to analyze the conflict. The same holds for unit propagation.

Otherwise, pure literals can be responsible for subsumed clauses to become conflicting. Then, the analysis is blocked as pure literals have no resolvable justification.

6.3 Development

The implementation went through several changes before reaching its current state. Starting from a basic SAT-solver with only unit propagation [10] and last UIP conflict analysis [12], I first naively implemented reduction.

A reduction in a clause was marked by the corresponding index on the stack. Consequently, every time a reduced literals truth value was computed, the entry on the stack had to be matched with the clause as the reduction could have been invalidated by backtracking. The advantage was that during backtracking the data structure was not updated following a common design philosophy of many sat solvers. Although the check has constant time, a literal's truth value is computed so frequently that 5 to 10 percent of the run time went into it. Therefore, this was later replaced by instead swapping reduced literals to the end of the clause, which avoids the costly check completely, while shifting the responsibility to the backtracking algorithm instead. The same was analogously changed for handling subsumption of clauses.

Also, the model was first implemented as a single stack containing decisions, propagations, reductions, and subsumptions as described by he calculus. However, subsumptions are in most cases very frequent and add an unnecessary burden to propagation and conflict analysis as both have to iterate over the model stack while filtering the subsumptions. Hence, subsumptions were separated from the rest into their own stack.

Next, the propagation initially alternated between unit propagation and subsumption resolution while iterating up the stack: In case of a decision or propagation literal, the clauses in the 2-watched-literal data structure were

updated and then the occurrence list was traversed with backward subsumption resolution. This, however, often lead to expensively found reductions becoming immediately irrelevant by following unit propagations. Hence, the change to nested loops where full unit propagation precedes the search for reductions and any additionally found propagation suspends the search for another round of unit propagation.

Furthermore, at the start Backward Subsumption Resolution mirrored UnitPropagation as it also had to handle when a candidate was violating the invariant of the 2-watched data structure due to a reduction. This was mostly redundant as most clauses where this was possible were already treated by UnitPropagation beforehand. By moving the responsibility to the reduction function, i.e., restoring the invariant whenever a reduction is found, a candidate is now always guaranteed to satisfy the invariant.

Unsurprisingly, the biggest changes saw the Backward Subsumption Resolution algorithm and the data structure for occurrence lists. Initially, the Partners for the reduction check would be naively taken from the union of the positive and negative occurrence lists of an atom occurring in the Candidate; preferably the one with the least occurrence, whereas the occurrence lists were simply arrays of clause pointers. Under this framework I had also implemented a check for pure literals as the traversal of the occurrence lists would determine whether the corresponding literal still occurred undefined in an unsatisfied clause.

The first big change was the addition of the FindPartner algorithm as described in section 5.2.4 based on Zhang's method [14]. However, using the existing data structure was still inefficient as every comparison for the intersections required dereferencing the clause pointer. Changing the occurrence sets to lists of clause numbers and accessing the clauses over the AllClauses map in the end immediately halved run times on my examples.

As intersections are a well studied topic in database systems, the next improvement was in part inspired by one such method [4]. I encoded the clause numbers into the block number-bit vector pairs to allow parallel intersections using AND instructions on the bit vectors. This brought on average another double speed up. Lastly, adding hash maps and using them after the first iteration again almost halved the time cost of the FindPartner function. With this change the algorithm spends only about a quarter of its time after the first iteration compared to about half before. Altogether, these changes lower ForwardSubsumption's percentage of the run time from about ninety percent to around forty.

To increase the pool of solvable problems for evaluation, I implemented further improvements to the base algorithm by adding standard features of

modern Sat solvers. First, conflict analysis learning the last UIP clause was replaced with learning the first UIP clause [12], which is further improved by recursive clause minimization [13] [7]. Lastly, Minisat's EVSIDS decision heuristic [6] replaces the original static variable ordering.

6.4 Evaluation

For the evaluation of the implementation, I use the SATLIB benchmark library [9] and especially the unsatisfiable problems from the Uniform Random-3-SAT benchmarks [2]. Their availability in several orders of difficulty allowed me to adjust testing with each step from the primitive first implementation to the finished version. Furthermore, as each size consists of one hundred unsatisfiable instances, the average results are stable against statistical outliers. Lastly, avoiding satisfiable instances mostly prevents unrepresentative comparisons due to lucky guesses.

Two versions of the implementation are evaluated. The first is the one described in chapter 5 with subsumption resolution. In the second Subsumption resolution is skipped. However, the now unused data structures for subsumption resolution remain. Both versions restart whenever the CNF formula's size doubles. In both versions, only subsumed clauses are removed using the backward subsumption resolution algorithm instead of forgetting clauses. While intended for the first version, this is done in the second as well to keep the size of the problem and consequently, the speed of propagation comparable.

The benchmarks are run on an Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz with eight gigabyte of memory with a five minute timeout.

6.4.1 First Experiment

In the first experiment, both versions use a static variable ordering and learn the last UIP clause. They are each run once on each of the one hundred problems in the uuf75-325, uuf125-538 and uuf150-645 sets and on one hundred problems of the one thousand in uuf100-430. The results are shown in Figure 6.1 and 6.2.

In the first set, the run times were too short to be comparable, and in the last set 63 runs reached the time out for the version with Subsumption resolution and 22 timed out for both versions. Figure 6.1 shows that on average subsumption resolution reduced the search depth by 20 to 25% and in some cases it almost halves depth. However, subsumption resolution adds

Problems	time %	conflicts %	decisions %
uuf75-325	-	71 (50-97)	72 (46-97)
uuf100-430	222 (116-422)	76 (55-109)	78 (56-106)
uuf125-538	362 (175-861)	77 (58-107)	79 (60-111)
uuf150-645	529 (324-867)	79 (70-92)	81 (72-94)

Figure 6.1: The results for the comparison of the two simple versions of the implementation: With and without Subsumption Resolution. Each column has the averaged relation as well as minimum and maximum of the first to the second version in percent for run time, number of conflicts and number of decisions.

Problems	Subsumptions %	Candidates %	Propagations
uuf75-325	56 (48-72)	18 (14-22)	28 (13-87)
uuf100-430	52 (46-60)	21 (15-32)	23 (7-39)
uuf125-538	51 (46-58)	24 (19-33)	20 (10-49)
uuf150-645	50 (47-55)	25 (21-31)	20 (10-35)

Figure 6.2: Some statistics for the occurrence of Subsumption Resolutions. The first column shows the proportions between reductions and subsumptions. The second shows the chance for each backward reduction check to find a reduction. The last column has the ratio of unit propagations to reductions.

considerable overhead, which also scales worse than the version without it as indicated by the increasing overhead with larger problem sizes.

Figure 6.2 contains information solely about the implementation with subsumption resolution. It shows that reductions and subsumptions occur with about the same frequency and on average checking a candidate with backward reduction has a one in four to five chance of finding a reduction. Further, there is a reduction for about every 25 unit propagations.

6.4.2 Second Experiment

In the second experiment, both versions use additionally recursively minimized 1UIP clause learning and the EVSIDS decision heuristic. This time each problem is run twice for each version with two different starting orderings – atom numbers in ascending and descending order. In the cases without timeouts, the two results for each problem are averaged, and the averages are used to compute the ratios. Due to the improvements, both versions can solve considerably larger problem instances.

Problems	time %	conflicts %	decisions %
uuf125-538	234 (111-400)	94 (59-130)	94 (57-129)
uuf150-645	302 (177-560)	96 (71-135)	96 (70-133)
uuf175-753	426 (179-886)	99 (67-138)	99 (67-138)
uuf200-860	631 (309-1132)	100 (70-122)	99 (69-122)
uuf225-960	867 (492-1400)	101 (82-124)	101 (82-123)

Figure 6.3: The results for the comparison of the two versions of the implementation: With and without Subsumption Resolution. Each column has the averaged relation as well as minimum and maximum of the first to the second version in percent for run time, number of conflicts and number of decisions.

Problems	Subsumptions %	Candidates %	Propagations
uuf125-538	26 (21-47)	14 (11-19)	163 (62-295)
uuf150-645	25 (21-44)	17 (13-22)	163 (79-306)
uuf175-753	25 (22-43)	20 (14-26)	167 (91-339)
uuf200-860	24 (21-42)	24 (21-31)	151 (94-221)
uuf225-960	23 (22-38)	26 (22-31)	151 (81-231)

Figure 6.4: Some statistics for the occurrence of Subsumption Resolutions. The first column shows the proportions between reductions and subsumptions. The second shows the chance for each backward reduction check to find a reduction. The last column has the ratio of unit propagations to reductions.

Timeouts occurred eleven times for the problem set uuf225-960. Figure 6.3 shows that while the overheads are still dominant, the effect of subsumption resolution seems to be completely gone. Apparently the standard improvements to CDCL somehow already include the effect of subsumption resolution. Figure 6.4 confirms that indeed the amount of reductions drastically decreases. While a candidates success rate stays roughly the same, reductions are now half as frequent compared to subsumptions and about six times rarer compared to unit propagations. With such a low occurrence it is not surprising that subsumption resolution has an unnoticeable impact on search depth.

6.5 Conclusion

In this thesis I have generalized unit propagation, a core concept of Sat solving. I have developed RCDCL, an extension of the CDCL calculus, that incorporates subsumption resolution into the search while keeping most aspects attributed to the efficiency of CDCL intact. Further, I have proven the correctness of the new calculus and implemented it into a working SAT solver. The evaluation of my implementation suggests that under a bare bones version of CDCL subsumption resolution indeed reduces search depth. However, modern techniques, such as learning 1UIP clauses and the VSIDS decisions heuristic, are preempting the effect.

6.5.1 Future Work

Although the evaluation results rather discourage further development of the calculus and its implementation, there are possible directions to continue.

First, during the evaluation I experienced that the overhead created by subsumption resolution restricted to in-processing was only about 10%. Comparing my implementation of propagation to existing pre- and in-processing algorithms might be interesting.

Also, the benchmark set for the evaluation was restricted to randomly generated problems. Problems commonly solved with SAT contain structures, where subsumption resolution might have an advantage. This was a mayor concern for me in the evaluation. Unfortunately, structured problems that are both unsatisfiable and solvable in a realistic time frame by my implementation are rare due to its large overhead .

Assuming such problem types exist or subsumption resolution were more effective, the overhead would still have to be dealt with. The main part of the subsumption resolution check, the FindPartner function, could be improved using multi-threading as each of the three intersections in an iteration can be computed individually. This could realistically cut the overhead of the reduction search in half.

Also, the current search is exhaustive, but only a fraction of checked candidate clauses results in subsumption resolutions, of which again only a fraction is actually involved in conflicts. An incomplete search using proper heuristics could avoid much of the unnecessary overhead.

Lastly, another approach could have been to use hidden literal elimination [8]. This is in a way restricting subsumption resolution to clauses of length two, while also using the clauses created by the transitive closure of the binary implication graph. This would have the added benefit of convergence, i.e., the order of the resolutions is irrelevant.

Bibliography

[1] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[2] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia*, pages 331–337, 1991.

[3] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[4] Bolin Ding and Arnd Christian Knig. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.

[5] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT'05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.

[6] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[7] Allen Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 141–146, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient cnf simplification based on binary implication graphs. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 201–215, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT, 2000.

[10] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001.

[11] Dennis Schwarz and Ching Hoo Tang. SPASSSATT. November 2011.

[12] João P. Marques Silva and Karem A. Sakallah. Grasp: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[13] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] Lintao Zhang. On subsumption removal and on-the-fly cnf simplification. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 482–489. Springer, 2005.