Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

Vollständiges Wort	Abkürzung
Conflict-driven clause learning	CDCL
Davis-Putnam	DP
Davis-Putnam-Logemann-Loveland	DPLL
Falsch	$\bot$
First Unique Implication Point	1UIP
Implementierung	Impl
Satisfiability Problem	SAT-Problem
Satisfiable	SAT
Unique Implication Point	UIP
Unsatisfiable	UNSAT
Wahr	$\top$

1 Einleitung

Das Boolean Satisfiability Problem (SAT-Problem, normalerweise nur SAT), auf Deutsch Erfüllbarkeitsproblem der Aussagenlogik, ist das erste Problem, welches durch den „Satz von Cook“ im Jahr 1971 als NP-vollständig bewiesen wurde [5]. Das SAT-Problem ist in der Praxis sehr wichtig, wie z.B. in der künstlichen Intelligenz [12] und Bounded Model Checking [2], weshalb trotz NP-Vollständigkeit viel Forschung in diesem Bereich betrieben wurde. Mitunter wurden mehrere SAT-Solver entwickelt, wie z.B. zChaff und MiniSAT.

Die Entwicklung von effizienten SAT-Solvern ist seit dem Jahr 2002 stetig angestiegen [24]. Für die Programmierung dieser Solver werden hauptsächlich objektorientierte Programmiersprachen verwendet. Im Gegensatz dazu ist der Anteil von SAT-Solvern, welche komplett in einer rein funktionalen Programmiersprache geschrieben wurden, sehr gering.

Im Rahmen dieser Bachelorarbeit wird der Frage nachgegangen, wie eine mögliche Umsetzung eines „Conflict-driven clause learning“ (CDCL) SAT-Solvers in einer rein funktionalen Programmiersprache aussehen könnte. Das Ziel dieser Arbeit ist eine Implementierung eines CDCL SAT-Solvers in Haskell und mögliche Vorschläge wie deren Implementierung verbessert werden kann.

Anhand einer Literaturrecherche werden bestehende Algorithmen untersucht, wofür verschiedene Arbeiten über CDCL und näherstehende Publikationen betrachtet werden. Die Literaturarbeit wurde gewählt, um bestehende Erforschungen in diesem Fachbereich zu erhalten.

In dieser Arbeit werden zuerst die verschiedenen Programmierparadigmen und die verwendete Programmiersprache Haskell in Kapitel 2 erläutert. Daraufhin werden SAT-Problem und verschiedene SAT-Solver in Kapitel 3 vorgestellt, wobei auch ein Einblick in die SAT-Competitions gewährt wird. Danach wird der Unterschied zwischen dem CDCL-Algorithmus und dem Davis-Putnam-Logemann-Loveland-Algo-

rithmus (DPLL) in Kapitel 4 erklärt. Das darauffolgenden Kapitel 5 beinhaltet eine Beschreibung für eine Umsetzung eines CDCL SAT-Solvers. Ein Vergleich der Implementierung mit anderen CDCL SAT-Solvern wird in Kapitel 6 durgeführt. Im letzten Teil der Arbeit (Kapitel 7) wird ein Fazit über die gewonnenen Erkenntnisse gezogen als auch ein Ausblick über Verbesserungsmöglichkeiten für eine bessere Effizienz des SAT-Solvers gegeben.

2 Funktionale Programmierung

In diesem Abschnitt werden die zwei verschiedenen Programmierparadigmen (imperative und deklarative Programmierung), funktionale Programmierung und die Programmiersprache Haskell kurz vorgestellt.

2.1 Imperative Programmierung

Folgende Definition wurde von der Universität Passau für imperative Programmierung verwendet: „Imperative Programme beschreiben Programmabläufe durch Operationen auf Zuständen“ [14]. Dies bedeutet dass der Ablauf des Programmes durch die Reihenfolge der Befehle maßgeblich für das erwartete Ergebnis ist. Sprachen, die zu diesem Paradigma gehören, sind z.B. Java, C++ oder C.
Die imperative Programmierung kann in weitere Paradigmen unterteilt werden, wie z.B. strukturierte Programmierung, objektoriertierte Programmierung und modulare Programmierung [1].

2.2 Deklarative Programmierung

Weiterführende Definition für deklarative Programmierung wurde von der Passauer Universität übernommen: „Deklarative Programme beschreiben Berechnungen durch eine Ein-/Ausgaberelation. Der Kontrollfluß ist dem Programmierer nicht explizit zugänglich; der Ablauf der Berechnung kann aber trotzdem durch den Programmaufbau beeinflußt werden.“ [14]. Sinngemäß bedeutet dies, dass nicht der Weg zum Ergebnis das Entscheidende ist, sondern die Spezifaktion des Problemes und der Ergebnisse selbst.
Die deklarative Programmierung wird wie die imperative Programmierung auch in verschiedene Unterkategorien eingeteilt. Diese sind z.B. logische Programmierung, Constraint Programmierung und funktionale Programmierung. Prolog, Lisp und SQL sind beispielhafte Programmiersprachen, die zu diesem Paradigma gehören [1].

Funktionale Programmierung

Wie im vorherigen Abschnitt erwähnt wurde, ist die funktionale Programmierung ein Teil der deklarativen Programmierparadigmen. Ein Programm in diesem Schema besteht hauptsächlich aus einer Zusammensetzung von Funktionen [14], verwendet unveränderliche Daten und limitiert Zustandsänderungen.
Die Programmierung mit funktionalen Sprachen bringt mehrere Vorteile. Beispiele für solche Vorteile sind „Lazy Evaluation“ und das explizierte Markieren von Seiteneffekten [21].
Folgende Beschreibung für Lazy Evaluation wurde von Bloss, Hudak und Young sinngemäß übernommen: Lazy Evaluation beurteilt einen Ausdruck erst, wenn der Wert absolut benötigt wird [4]. Dies bedeutet, dass eine Parameterübergabe an eine andere Funktion eine Evaluierung nicht auslöst. Der Wert muss explizit für die Berechnung eines neuen Wertes aufgerufen werden, damit dieser evaluiert wird.
Eine Definition für Seiteneffekte wurde von Nokie übernommen: Seiteneffekte sind Änderungen im Programmzustand. Diese können z.B. durch Funktionsaufrufe oder der Veränderung einer globalen Variablen ausgelöst werden [22]. Dadurch haben Seiteneffekte Auswirkung auf spätere Funktionsaufrufe, weshalb das explizierte Makieren von Seiteneffekten bestimmte Fehler leichter ausschließen lassen kann.

Haskell

Haskell ist eine der weitverbreitesten funktionalen Sprachen, die 1988 ihren Namen im „Yale Meeting“ erhielt. Der Name kam vom Mathematiker Haskell B. Curry, wessen Arbeit einer der Anstöße zur Entwicklung von Haskell geführt hat. Im Jahr 1987 begann der Haskell Design Prozess in der „Functional Programming and Computer Architecture Conference“ (FPCA) und am 01. April 1990 wurde der „Haskell version 1.0 report“ veröffentlicht. Über die Jahre entwickelte sich die Sprache weiter und es wurde im Februar 1999 der „Haskell 98 Report“ veröffentlicht, wobei eine Revision im Dezember 2002 veröffentlicht wurde [10]. Mit der Veröffentlichung des „Haskell 2010 Language Report“ wurde eine wichtige Änderung für neue Revisionen beschlossen. Jedes Jahr sollte mindenstens eine neue Revision veröffentlicht werden, die eine kleine Anzahl an Änderungen und Erweiterungen beinhaltet [17].

Die Sprache Haskell wurde für diese Arbeit ausgewählt, sodass die Implementierung in Lehrveranstaltungen verwendet werden kann, die oft Haskell nutzen.
Im Folgenden werden Beispielfunktionen in Haskell gezeigt, die die vorgeführten Vorteile im Abschnitt „Funktionale Programmierung“ darstellt.

Listing 2.1: Ein einfaches „Hello World“-Programm

```
1 main :: IO ()
2 make = putStrLn "Hello World"
```

Wie in Zeile 1 im Programmbeispiel 2.1 gezeigt wird, werden Seiteneffekte in Haskell immer durch IO markiert. Wenn eine Funktiondeklaration keine IO beinhaltet, so ist die Funktion seiteneffektfrei.

Listing 2.2: Beispiele für Lazy Evaluation

```
1 lazy :: Int -> Int -> Int
2 lazy x y = x
3
4 lazy2 :: Int -> Int -> Int
5 lazy2 x y = if x > 2 then y else x
```

Im Programmbeispiel 2.2 würde die „Eager Evaluation“ (das Gegenteil von Lazy Evaluation) die Parameter x und y in Zeile 2 evaluieren. Lazy Evaluation hingegen evaluiert nur den Parameter x, da y nicht weiter verwendet wird in der Funktion. In der Beispielsfunktion „lazy2“ wird der Parameter y in Zeile 5 nur evaluiert, wenn x größer als 2 ist.

3 SAT

Dieses Kapitel gibt eine kurze Einführung zum Thema SAT und einen Einblick in die SAT-Competitions. Des Weiteren werden die bekannten SAT-Solver zChaff und MiniSAT vorgestellt.

3.1 SAT-Problem

Die SAT Association definiert das SAT-Problem als ein Problem, in dem bestimmt werden muss, ob für ein aussagenlogisches Problem boolesche Wertzuweisungen existieren, die dieses dann zu einer wahren Aussage evaluiert [23]. Stephen A. Cook hat die NP-Vollständigkeit des SAT-Problems 1971 in seiner Publikation bewiesen [5], welche 1973 nochmals von Leonid A. Levin nachgewiesen wurde [15]. Deshalb wird der Beweis oft auch „Satz von Cook-Levin“ genannt. Dieser Beweis führte dazu, dass das SAT-Problem als erstes Problem die NP-Vollständigkeit nachgewiesen wurde.
Jedes aussagenlogische Problem kann in „Konjunktiver Normalform“ oder zu Englisch „Conjunctive Normal Form“ (CNF) umgeschrieben werden. CNF ist eine Konjunktion ($\bigwedge$) von Disjunktionen ($\bigvee$).
Ein aussagenlogisches Problem in CNF besteht aus folgenden Elementen:

- Literale
- Klauseln
- Formeln

In dieser Arbeit ist ein Literal eine Variable, welches mit Wahr ($\top$) oder Falsch ($\bot$) belegt werden kann und negiert ($\neg$) sein kann. Die Belegung eines Literals kann folgendermaßen aussehen:

$$x_1 \equiv \top,\ x_2 \equiv \bot \tag{3.1}$$

Eine Klausel besteht aus einer Menge von disjunkten Literalen, während eine Formel eine Menge von Klauseln ist.

Folgende Beispiele zeigen Formeln, die in CNF oder nicht in CNF sind.

$$x_1 \wedge (x_2 \vee x_3) \tag{3.2}$$

$$x_1 \wedge \neg x_2 \tag{3.3}$$

$$x_1 \vee x_2 \tag{3.4}$$

$$x_1 \wedge (x_2 \wedge \neg x_3) \tag{3.5}$$

$$x_1 \vee (x_2 \wedge \neg x_3) \tag{3.6}$$

Die Beispiele 3.2, 3.3 und 3.4 sind in CNF Form, während 3.5 und 3.6 nicht in CNF sind. Der Grund hierfür liegt daran, dass die Literale innerhalb der Klausel mit $\vee$ verknüpft sind. Das Beispiel 3.3 ist in CNF, da die Literale x_1 und x_2 in disjunktiver Form sind. Diese Probleme können mit zwei Ergebnissen beurteilt werden. Als Ergebnis können die Probleme entweder mit Satisfiable (SAT) oder Unsatisfiable (UNSAT) evaluiert werden.

$$\neg x_1 \vee x_2 \tag{3.7}$$

$$x_1 \vee x_2 \tag{3.8}$$

Die Literalbelegung aus 3.1 würde die Klausel in 3.7 zu falsch evaluieren. Die Klausel in 3.8 dahingegen wird zu wahr evaluiert.

Das SAT-Problem findet sich in vielen industriellen Bereichen wieder, die sich mit der Informatik beschäftigen. Darunter zählen z.B. Bounded Model Checking[2], künstliche Intelligenz [12] und Theorembeweise [23].

3.2 SAT-Competition

Die internationale SAT-Competition findet seit dem Jahr 2002 statt. Der Wettbewerb wurde eingeführt, um neue SAT-Solver vorzustellen und Benchmarks zu finden, die nicht einfach zu lösen sind. Dabei werden die Solver auch mit SAT-Solvern verglichen, die den Stand der Technik darstellen. Unter anderem existieren im Wettbewerb

auch unterschiedliche Disziplinen, in denen sich die Solver messen können [24].

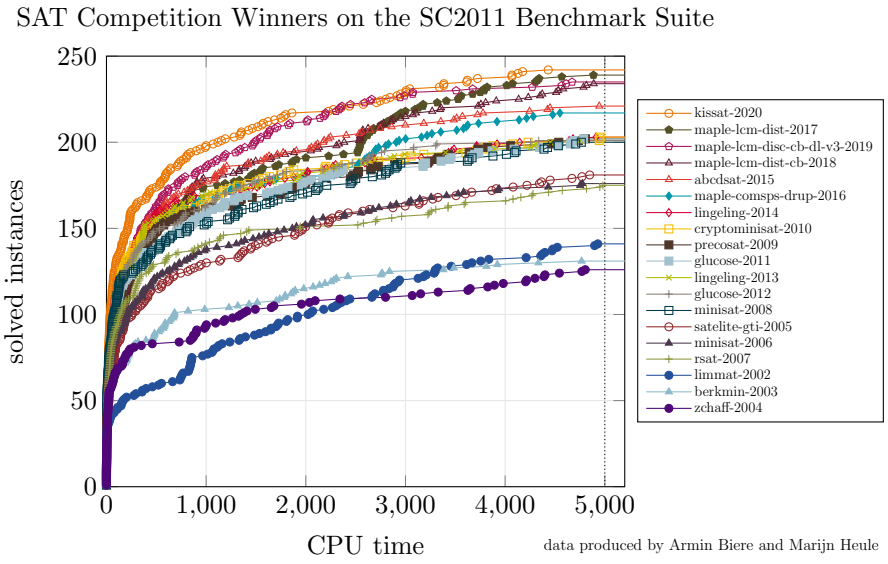

Abbildung 3.1: Benchmark 2011
Bildquelle: http://fmv.jku.at/kissat/winners-2011.pdf

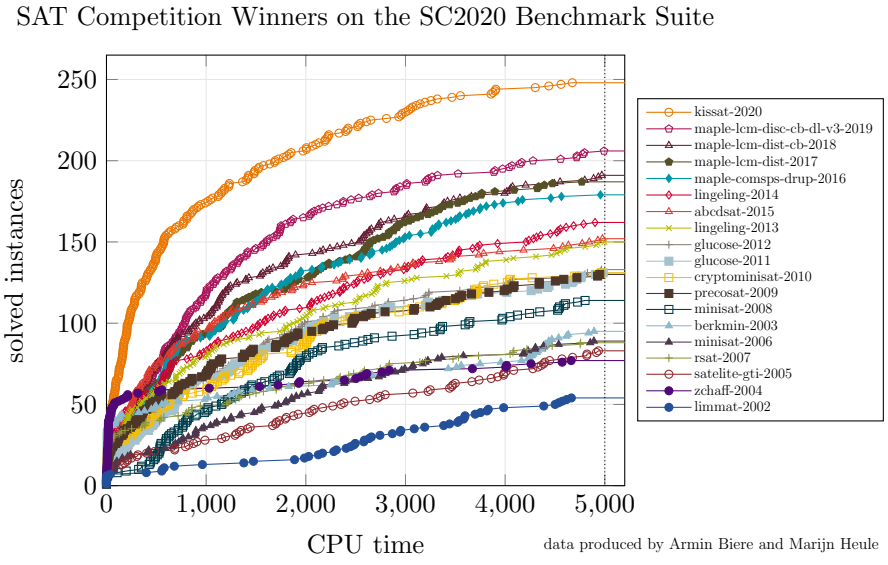

Abbildung 3.2: Benchmark 2020
Bildquelle: http://fmv.jku.at/kissat/winners-2020.pdf

Die y-Achse der Abbildungen 3.1 und 3.2 zeigt die Anzahl der gelösten SAT-Probleme, während die x-Achse die CPU Zeit darstellt. Die Legende stellt die Gewinner der SAT-Competition von 2002 bis 2020 dar. Verglichen mit der Abbildung 3.1 wird ersichtlich, dass die Benchmarks aus Abbildung 3.2 komplexer wurden, was durch die Ausweitung der Graphen wiedergespiegelt wird. Daraus kann geschlossen werden, dass die Solver mit den Jahren einen höhere Leistungsgrad erzielt haben [3].

3.3 SAT-Solver

Es existieren viele SAT-Solver, die gute Leistungen vollbringen beim Lösen von SAT-Problemen. In den folgenden Abschnitten werden zwei Solver vorgestellt, welche an einer SAT-Competition teilgenommen und diese auch zu ihrer Zeit gewonnen haben. Die zwei SAT-Solver, die vorgestellt werden, sind zChaff und MiniSAT.

3.3.1 zChaff

Der SAT-Solver zChaff basiert auf den Chaff-Algorithmus und wurde an der Princeton Universität entwickelt und veröffentlicht. Der Chaff-Solver verwendet einen optimierten „Boolean Constrainst Propagation"-Algorithmus (BCP, siehe 5.3.2) mit „Watched Literals" (siehe 5.2) und „Variable State Independent Decaying Sum" (VSIDS, siehe 5.3.1) für die Entscheidungsheuristik. Des Weiteren wendet zChaff „Restarts" und „Clause Deletion" an [20]. Viele von diesen Ideen wurden später für den 2003 entwickelten MiniSAT übernommen [8].
zChaff hat 2004 im „Industrial Track" die „ALL (SAT+UNSAT)" und „UNSAT" Kategorien gewonnen [24].

3.3.2 MiniSAT

MiniSAT wurde von Niklas Eén und Niklas Sörensson an der Chalmers University of Technology entwickelt. Hintergrund für die Entwicklung des Solvers ist die Veröffentlichung eines zugänglichen, minimalistischen CDCL-SAT-Solvers, der Watched Literals und Dynamic Variable Ordering (VSIDS) verwendet [8]. Durch dieses Konzept sind viele SAT-Solver entstanden, die MiniSAT als Basis für ihre Entwicklung verwendet haben.
MiniSAT gewann den ersten Platz im SAT-Race 2006 [25].

3.4 DIMACS Format

Damit SAT-Solver ein generelles Dateiformat einlesen könnnen, um aussagenlogische Probleme zu lösen, wurde das DIMACS Format eingeführt.

Listing 3.1: DIMACS Format

```
1 c simpleExample.cnf
2 c
3 p cnf 3 2
4 1 2 -3 0
5 c Kommentare können auch hier sein
6 -2 -1 0
```

Die Spezifikationen des SAT-Problems werden zeilenweise definiert. Kommentare werden mit einem kleinem c symbolisiert wie z.B. in Zeile 1 und 5 im Beispiel, während kleine p's die Spezifikationen des SAT-Problemes zeigen. Das Wort nach dem p deutet in welcher Form ein Problem dargestellt ist. Dies kann entwender CNF oder „Disjunktive Normalform“ (DNF) sein. Der erste Integer in Zeile 3 steht für die Anzahl der Literale und der Zweite für die Anzahl der Klauseln. Wenn nach der p-Zeile ein Integer oder ein Minus die Zeile anführt, so beginnt eine Klausel. Integer in diesen Zeilen können entweder positive oder negierte Literale darstellen. Wenn eine 0 eingelesen wird, ist die Klausel vollständig und es wird eine neue Zeile eingelesen.
Die decodierte Formel für das obige Beispiel sieht folgendermaßen aus:

$$(1 \vee 2 \vee -3) \wedge (-2 \vee -1)$$

4 Algorithmen

In diesem Kapitel werden der DPLL-Algorithmus und CDCL-Algorithmus vorgestellt. Anhand einer Literaturarbeit werden die Erweiterung erläutert, die das CDCL-Verfahren performanter im Gegensatz zu DPLL machen.

Die in der Arbeit vorgestellten Algorithmen sind bewährte Methoden, um SAT-Probleme zu lösen. Die Regeln des DPLL-Algorithmus wurden sinngemäß aus den Publikation von Davis, Putnam, Logemann und Loveland übernohmen [7, 6]. Die Definitionen, die im Kapitel 4.2 vorkommen, wurden sinngemäß aus dem Buch von Kroening und Strichman [13] übernommen.

4.1 DPLL-Algorithmus

Martin Davis und Hilary Putnam entwickelten 1960 den Davis-Putnam-Algorithmus (DP) [7], der als Basis für das 1962 entwickelte DPLL-Verfahren verwendet wird. Der Algorithmus wurde von Martin Davis, George Logemann und Donald Loveland präsentiert und besitzt drei Regeln [6].
Die erste Regel ist die Eliminierung von Klauseln mit nur einem Literal [7]. Klauseln, die nur ein Literal besitzen, werden auch atomare Klausel genannt. Mit dieser Regel wurden vier Teilregeln eingeführt. Diese sind folgende:

1. Existieren zwei atomare Klauseln mit gegenteiligen Literalen, wie z.B. $\{p\}$ und$\{\neg p\}$, so ist die Formel F UNSAT.

2. Existiert eine atomare Klausel mit nur einem positiven Literal p, so können alle Klauseln, die ein p beinhalten, gelöscht werden. Negierte p's werden dahingegen aus allen Klauseln gelöscht.

3. Die dritte Teilregel ist wie die zweite Teilregel, jedoch mit umgekehrten Vorzeichen.

4. Die Formel ist SAT, wenn die Formel F nach iterativer Anwendung der Regeln leer wird.

Die zweite Regel ist die „Affirmative-Negative Rule“ [7]. Diese Regel ist ähnlich zu den Teilregeln 2 und 3 aus der ersten Regel 2. Anstelle von atomaren Klauseln wird hier eine Entscheidung für ein Literal in einer Klausel getroffen. Wenn das Literal p mit $\top$ belegt wird, so werden alle Klauseln mit einem positiven p gelöscht, während alle negierten p's aus den Klauseln gelöscht werden. Dies geschieht auch umgekehrt mit einer Literalbelegung von $\bot$.
Als dritte Regel wurde die Teilungs Regel vorgestellt [6]. In dieser Regel wird eine Formel F in die Form $(A \vee p) \wedge (B \vee \neg p) \wedge R$ aufgeteilt, wobei p nicht in A, B und R vorhanden ist. A und B stellen hierbei eine Menge von anderen Literalen dar, während R eine Menge von anderen Klauseln in F darstellt. F wird UNSAT, wenn für die Belegungen $p \equiv \bot$ und $A \wedge R$ und $p \equiv \top$ und $B \wedge R$ die Klauseln in der Formel Inkonsistenz aufzeigen.
Ein Beispiel wie die Regeln 1 und 2 funktionieren, wird an folgendem Problem gezeigt:

$$F = x_1 \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_4 \vee x_5) \tag{4.1}$$

$$Setze\, x_1 \equiv \top$$

$$\Leftrightarrow \neg x_2 \wedge (x_3 \vee x_4) \wedge (x_4 \vee x_5) \tag{4.2}$$

$$Setze\, x_2 \equiv \bot$$

$$\Leftrightarrow (x_3 \vee x_4) \wedge (x_4 \vee x_5) \tag{4.3}$$

$$Setze\, x_4 \equiv \top$$

$$\Rightarrow SAT\, mit\, Belegung:\ x_1 \equiv \top, x_2 \equiv \bot, x_4 \equiv \top \tag{4.4}$$

Die zweite Teilregel der ersten Regel wird im ersten Schritt angewendet. Diese löscht alle Klauseln, die eine positive x_1 besitzen, und alle negierten x_1 aus allen Klauseln. Dadurch kommt die Formel $\neg x_2 \wedge (x_3 \vee x_4) \wedge (x_4 \vee x_5)$ im zweiten Schritt zustande. Nach der Anwendung der dritten Teilregel für das Literal x_2 kommt die Formel $(x_3 \vee x_4) \wedge (x_4 \vee x_5)$ als Ergebnis heraus. Mit dem Einsetzen von $x_4 = \top$ wird die

zweite Regel benutzt, wodurch die leere Menge entsteht. Dadurch wird die gesamte Formel mit SAT evaluiert.
Wenn die Formel F jedoch noch eine atomare Klausel ($\neg x_1$) besitzen würde, würde F zu UNSAT evaluiert werden. Der Grund hierfür ist, dass die erste Teilregel bei zwei atomaren Klauseln mit unterschiedlichen Vorzeichen und gleichen Literalen die Formel als UNSAT beurteilt.

4.2 CDCL-Algorithmus

CDCL SAT-Solver basieren auf dem DPLL-Algorithmus und verwenden zusätzlich zu den Regeln mehrere Optimierungen und einen Algorithmus, die dem CDCL seinem Namen verdankt. CDCL wurde in Publikationen von Marques-Silva und Sakallah 1996 [18] und 1999 [19] und Bayardo und Schrag 1997 [11] vorgeschlagen. Der CDCL-Algorithmus wird immer dann angewendet, wenn nach der Anwendung des Entscheidungsalgorithmus (siehe 5.3.1) und Boolean Constraint Propagation (BCP, siehe 5.3.2) ein Konflikt auftritt. Ein Konflikt ensteht, wenn eine Zuweisung von Literalen bei einer Klausel keine logische wahre Aussage ergibt. Das folgende Beispiel soll diesen Fall zeigen:

$$F = (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \tag{4.5}$$

$$Setze\, x_1 \equiv \top\, (Entscheidung)$$

$$\Leftrightarrow \neg x_2 \wedge x_2 \tag{4.6}$$

$$Setze\, x_2 \equiv \bot$$

$$\Rightarrow Konflikt\, bei\, Klausel\, (\neg x_1 \vee x_2)$$

Die Klausel ($\neg x_1 \vee x_2$) hat einen Konflikt für die Belegungen $x_1 = \top$ und $x_2 = \bot$ verursacht. In der Publikation von Marques-Silva und Sakallah [19] wird die Verwendung eines Implikationsgraphen vorgeschlagen, um diesen Konflikt zu lösen. Ein Implikationsgraph ist ein gerichteter, azyklischer Graph, der alle derzeitigen Literalbelegungen eines SAT-Problemes darstellt. Hierfür stellen die Wurzelknoten des Graphen die Entscheidungen dar, während andere Knoten Literalbelegungen durch

Unitpropagation (siehe 5.3.2) abbilden. Die Kanten stellen Klauseln dar, die zur Belegung von Literalen geführt haben. Des Weiteren zeigen Knoten auch zu welchem Entscheidungslevel (Level) ein Literal belegt wurde. Beim Start des CDCL-Algorithmus fängt das Level mit dem Integer 0 an und steigt immer um 1, wenn eine Entscheidung gefällt werden muss. Konflikte werden im Graphen, wie die Belegungen, auch als Knoten dargestellt. Sollte ein Konfliktknoten im Level 0 auftreten, so ist das SAT-Problem UNSAT.
Für das Beispiel in Formel 4.5 würde der Graph folgendermaßen aussehen:

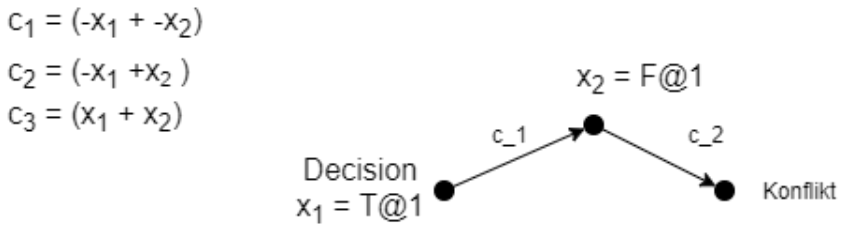

Abbildung 4.1: Beispiel eines Implikationsgraphen
Die + Operatoren stellen $\vee$ dar. Das F steht in den Abbildung für $\bot$

Der Graph in Abbildung 4.1 ist gleichzeitig auch ein partieller Implikationsgraph, da dieser alle relevanten Entscheidungen und Literalbelegungen zum Konflikt im zurzeitigen Level zeigt. Um die Konfliktklausel zu lernen wird Resolution [13] verwendet.
Hierbei ist die Resolution folgende Regel:

$$\frac{x_1 \vee x_{...} \vee x_n \vee p \qquad y_1 \vee y_{...} \vee y_n \vee -p}{x_1 \vee x_{...} \vee x_n \vee y_1 \vee y_{...} \vee y_n} \tag{4.7}$$

Die Resolution benötigt mindestens zwei Klauseln, die ein polarisierendes Literal beinhalten. In der Formel 4.7 wird dies durch die zwei Klauseln oberhalb des Striches dargestellt. Diese Klauseln werden auch resolving clauses (RC) genannt. Durch die Anwendung der Regel wird das ausgewählte polarisierende Literal aus den Klauseln gelöscht und die Klauseln werden zu der Klausel im unteren Teil der Formel vereinigt. Das Literal p ist hierbei eine resolution variable, die nach der Resolution ihre Literalbelegung verliert. Die berechnete Klausel wird als resolvent clause bezeichnet und kann als Klausel für die nächste Resolution verwendet werden oder zur Formel als gelernte Konfliktklausel hinzugefügt werden. Die Resolution kann nicht mehr angewendet werden, wenn keine Klauseln mehr existieren, die jeweils ein Literal p mit polarisierenden Vorzeichen besitzen.
Die Resolutionschritte gehen einen Implikationsgraphen, der von links nach rechts entwickelt wird, von rechts nach links ab. Dabei gibt es verschiedene Stopkriterien, die verwendet werden können, um die Konfliktklausel zu lernen. In dieser Arbeit wird

„First Unique Implication Point“ (1UIP) [13] als Stopkriterium für die Konfliktanalyse verwendet. 1UIP ist ein „Unique Implication Point“ (UIP), der dem Konfliktknoten am nähesten ist. Dies liegt daran, dass die Entscheidung an diesem Knotenpunkt einen Konflikt augelöst hat. Solange die Konfliktklausel nach dem „Backjumping“ nur ein unbelegtes Literal besitz, können die SAT-Solver in diesem Prozess bis zum zweithöchsten Level zurückspringen. Dadurch werden alle Literalbelegungen vom höchsten bis zum zweithöchsten + 1 gefundenem Level gelöscht. Dieser Prozess wird auch Backjumping genannt, da hierbei über mehrere Level zurückgesprungen werden kann, während beim Backtracking immer nur ein Level zurückgesprungen wird.

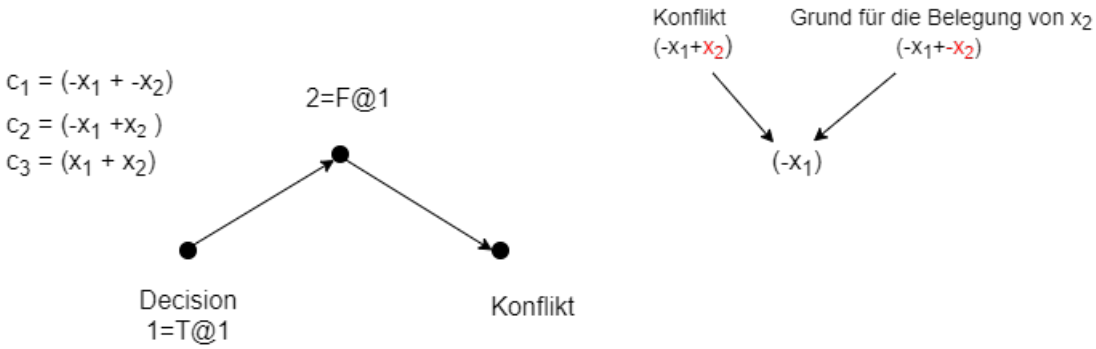

Abbildung 4.2: Resolution
Die Grafik illustriert den Ablauf der Resolution für das Beispiel in der Abbildung 4.1

In der Abbildung 4.2 werden die Klauseln c_1 und c_2 für die Resolution verwendet, da die Klausel c_2 den Konflikt verursacht hat und die Klausel c_1 für die Belegung der Variable x_2 verantwortlicht ist. Als Ergebnis kommt die neue Klausel $(\neg x_1)$ heraus und wird zur Formel 4.5 hinzugefügt. Gleichzeitig werden alle Belegungen, die auf dem Entscheidungslevel 1 getroffen wurden, gelöscht und das Level wird auf 0 zurückgesetzt. Danach wird der BCP-Prozess fortgeführt und eine Lösung mit den Belegungen $x_1 \equiv \bot@0$ und $x_2 \equiv \top@0$ wird gefunden.

4.3 Unterschied zwischen DPLL und CDCL

Der Unterschied zwischen DPLL und CDCL ist das Konzept des Backjumpings und das Lernen von Konfliktklauseln. Während der DPLL-Algorithmus durch Backtracking Konflikte löst, verwendet CDCL diese Konzepte, um schneller Ergebnisse zu einem SAT-Problem zu finden. Die folgenden Grafiken sollen zeigen, was das Problem vom Backtracking ist.

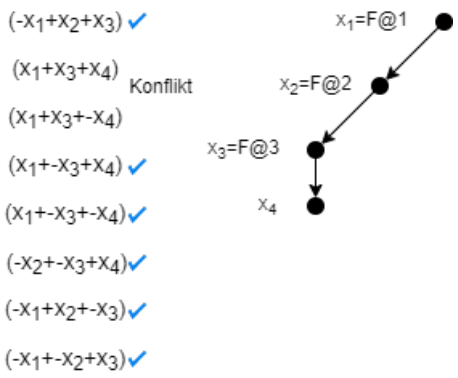

Abbildung 4.3: DPLL 1

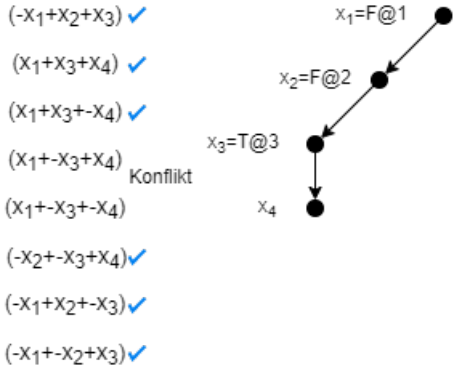

Abbildung 4.4: DPLL 2

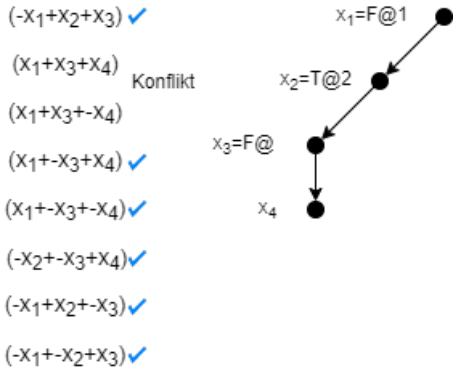

Abbildung 4.5: DPLL 3

Der DPLL-Algorithmus findet in der Grafik 4.3 einen Konflikt für die Belegungen $x_1, x_2, x_3 \equiv \bot$. Die Belegung von x_4 ist hierbei irrelevant, da jede beliebige Belegung

zu einem Konflikt führt. Im nächsten Schritt (Abbildung 4.4) würde der Algorithmus dann $x_3 \equiv \top$ setzen. Diese Belegung ändert jedoch nichts am Ergebnis und führt zum nächsten Backtracking-Schritt in der Grafik 4.5. In diesem Schritt bleibt der Konflikt jedoch erhalten und es würde so lange weitergehen, bis der Algorithmus eine Lösung mit der Belegung $x_1 \equiv \top, x_2 \equiv \top, x_3 \equiv \top$und $x_4 \equiv \top$ findet.

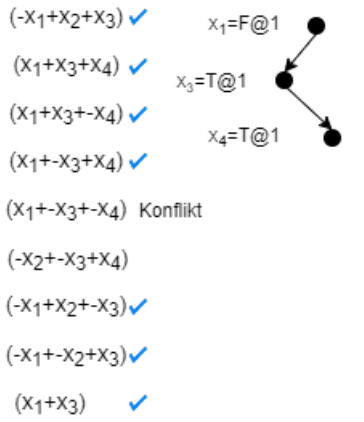

Abbildung 4.6: CDCL 1

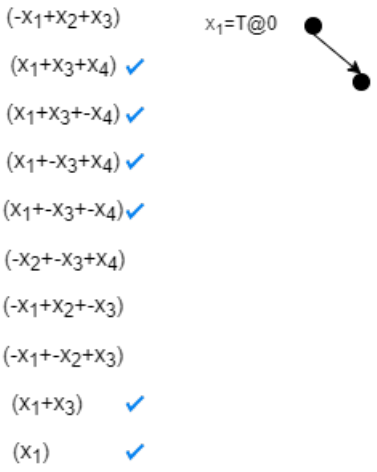

Abbildung 4.7: CDCL 2

Der CDCL-Algorithmus überspringt dahingegen die Schritte in Abbildung 4.4 und 4.5. Der Grund hierfür ist, dass der Algorithmus durch Resolution in der Abbildung 4.3 die Klausel $(x_1 \vee x_3)$(siehe Abbildung 4.6) lernt. Dadurch springt der Algorithmus zum Level 1 zurück und belegt das Literal $x_3 \equiv \top$. Die Belegung von $x_4 \equiv \top$ führt jedoch in der Klausel $(x_1 \vee \neg x_3 \vee \neg x_4)$ zum nächsten Konflikt. Durch Resolution mit den Klauseln $(x_1 \vee \neg x_3 \vee x_4)$ und $(x_1 \vee x_3)$ wird die Klausel (x_1) (siehe Abbildung 4.7) gelernt und der Algorithmus springt zum Level 0 zurück. Es wird ersichtlich, dass der CDCL-Prozess relativ viele Schritte überspringt, die der DPLL-Prozess berechnet hätte.

5 Implementierung des SAT-Solvers

In diesem Kapitel wird die Implementierung des SAT-Solvers diskutiert. Hierbei werden zunächsts die notwendigen Datenstrukturen beschrieben und danach werden die Algorithmen und der Aufbau des Programmes dargestellt.

Die wichtigsten Algorithmen in der Implementierung basieren auf Konzepten, die von zChaff und MiniSAT verwendet werden.

5.1 Programmaufbau

Die Implementierung wurde in mehreren Module aufgeteilt. Folgende Module wurden für das Programm erstellt:

- Algorithm
- Types
- Unitpropagation
- Decisionalalgorithm
- Conflict
- MapLogic
- CDCLFilereader

Im Algorithm-Modul befindet sich die Funktionen, die den Algorithmus starten, die Funktionen in den anderen Modulen aufruft und die Interpretierung der Formel basierend auf den Belegungen der Literale. Im Types-Modul befinden sich die Definitionen der Datenstrukturen und Hilfsfunktionen, die auf entsprechende Daten zugreifen. Im Unitpropagation-Modul finden sich relevante Funktionen wieder, die für das BCP verantwortlich sind. Dazu zählen auch die Funktionen „Unit Resolution“ und „Unit Subsumption“. Im Decisionalalgorithm-Modul ist die Implementierung der

Heuristik und die Auswahl der Literale, die belegt werden soll. Das Conflict-Modul beinhaltet den Algorithmus zur Analyse eines Konflikts und das Lernen neuer Klauseln. Das MapLogic-Modul ist verantwortlicht für die Aufrechterhaltung der Map, die die Daten für die Level und Literalbelegung enthält. Das CDCLFilereader-Modul liest eine cnf-Datei ein und ruft die Startfunktion im Algorithm-Modul auf und druckt am Ende das Ergebnis in der Konsole aus.

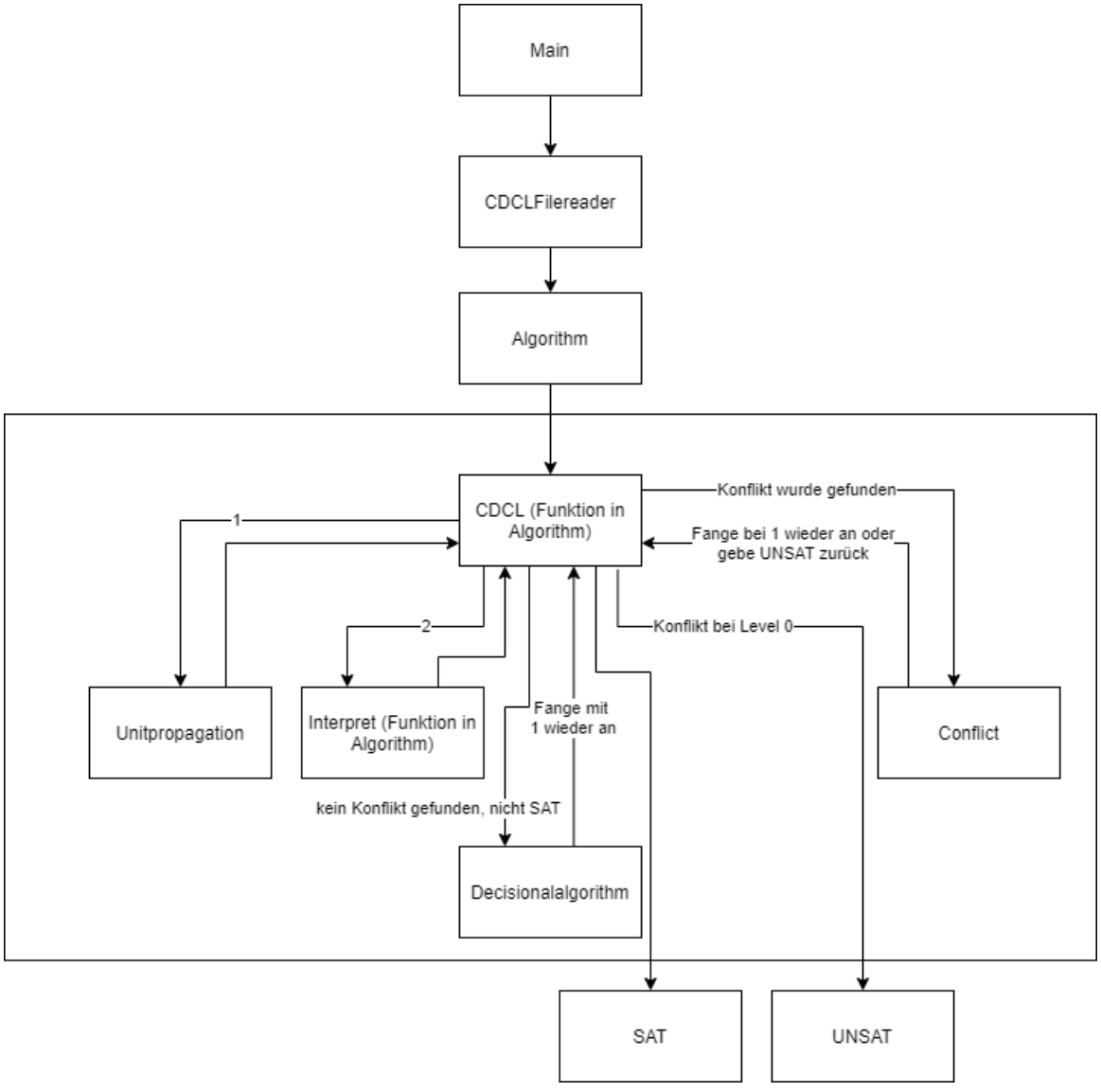

Abbildung 5.1: Grober Programmaufbau

5.2 Datenstrukturen

In der Implementierung werden verschiedene Datenstrukturen verwendet.

Listing 5.1: Zentrale Datenstrukturen. Siehe Anhang für weitere Strukturen

```haskell
-- | Literal definiert als Integer
newtype Literal = Lit Integer

-- | Klausel definiert als eine Liste von Literal
type Clause = [Literal]

-- | Tuple aus zwei Klauseln
--   Erste Klausel wird reduziert durch Unitresolution
--   Zweite Klausel ist die originale Form der Klausel
type ReducedClauseAndOGClause = (Clause, Clause)

-- | ClauseList definiert als Liste von ReducedClauseAndOGClause
type ClauseList = [ReducedClauseAndOGClause]

data Reason = Decision | -- Literal wurde durch Entscheidung belegt
                         Reason Clause -- Literal wurde durch
                             Unitpropagation belegt

data BoolVal = BFalse | -- Literalbelegung mit Falsch
                        BTrue | -- Literalbelegung mit Wahr
                        BNothing -- keine Belegung des Literals

-- | Assoziert alle Literale eines Problems mit einer Aktivität
--   Notwendig für Heuristik
type ActivityMap = Map.Map Literal Activity

-- | Eine Map mit Level als Key und TupleClauseList als Value.
--   Assoziert alle Literalbelegungen in der Reihenfolge, in der sie
--   belegt wurden auf dem entsprechenden Level. Notwendig für
    Konfliktanalyse
type MappedTupleList = Map.Map Level TupleClauseList
```

Zum Zeitpunkt der Implementierung wurde entschieden keine Watched Literals zu implementieren. Watched Literals verwendet zwei Zeiger, die auf verschiedene Literale innerhalb einer Klausel zeigen. Wenn ein Literal mit einem Zeiger belegt wird, wird überprüft, ob die Belegung die Klausel SAT macht. Bei einem Ergebnis mit SAT bleibt der Zeiger auf dem Literal, während bei einem UNSAT der Zeiger zum nächsten unbelegten Literal wandert. Falls der Zeiger kein unbelegtes Literal findet, bleibt der Zeiger auf seiner Position. Dabei können die Klauseln zwei verschiedene Zustände erhalten. Wenn einer der Zeiger auf ein unbelegtes Literal zeigt, so ist die Klausel UNIT. Ist dies nicht der Fall, so ist die Klausel leer und die Konfliktanalyse wird gestartet.

Das Konzept der Watched Literals ist eine komplexe Implementierung in Haskell, da die Reihenfolge der Literale innerhalb der Klauseln geändert werden muss.

Der Grund hierfür ist, das Haskell nicht direkt in konstanter Zeit auf die einzelnen Listenelemente zugreifen kann und kein destruktives Update möglich ist. Aus

diesem Grund wird stattdessen ReducedClauseAndOGClause verwendet, wobei das erste Element eine veränderliche Klausel darstellt, während das zweite Element eine unveränderte Klausel ist.

Für die Datenstruktur für die Heuristik und der Literalbelegung wurde das Haskellpaket Data.Map.Strict ausgewählt. Der Grund für die Auswahl von Maps ist die Schnelligkeit, in der Dateneinträge gefunden werden können. Bei einer Map ist die Komplexität für so eine Operation $O(log\, n)$, während bei einer Liste eine Komplexität von $O(n)$ besteht.

Ein Implikationsgraph wird in der Implementierung nicht explizit verwendet, da die Resolution ohne diese auskommt. Die Klauseln, die für die Resolution verwendet werden, kommen von der MappedTupleList.

Listing 5.2: Beispiele der Datenstrukturen aus dem Programmlisting 5.1

```
1  -- | Literal
2  Lit 1
3
4  -- | Clause
5  [Lit 1, Lit 2]
6
7  -- | ReducedClauseAndOGClause
8  ([Lit 1], [Lit 1, Lit 2])
9
10 -- | ClauseList
11 [([Lit 1], [Lit 1, Lit 2]), ([Lit 1, Lit 3], [Lit 1, Lit 3])]
12
13 -- | ActivityList
14 Map.fromList [(Lit 1, Activity 1),(Lit 2, Activity 1),(Lit 3, Activity
       2)]
15
16 -- | MappedTupleList. Abstände wurden hier ein
17 Map.fromList [(Level 1, [ ((Literal 1, BFalse), Decision), ((Literal 2,
       BTRue), Reason [Lit 1, Lit 2]) ])]
```

5.3 Algorithmen

5.3.1 Entscheidungsalgorithmus

Im Entscheidungsalgorithmus wird VSIDS verwendet. Bei dieser Heuristik werden alle negativen und positiven Literale zusammengezählt. Diese berechneten Werte werden in der Implementierung als Activity bezeichnet und werden in der Activity-Map zu den entsprechenden Literalen zugeordnet. Über die Zeit wird der Wert der

Activity durch einen konstanten Faktor reduziert, wobei die Activity durch Konflikte um 1 erhöht werden kann. Dabei werden nur die Activities erhöht, die in der gelernten Klausel vorkommen.
Mit VSIDS haben somit Literale, die öfter in Konflikten vorkommen, einen höheren Stellwert und werden eher zuerst belegt als Literale, die nicht so oft vorkommen.
In der Implementierung wird die erste höchste Activity gesucht. Danach wird dann die erste Klausel mit dieser Activity gesucht und das Literal wird so belegt, dass die Belegung insgesamt zu einem logischen Falsch in der Klausel belegt wird. Das folgende SAT-Problem illustriert dies in einem kurzem Beispiel.

$$F = (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \tag{5.1}$$

$$Setze\, x_1 \equiv \top$$

$$\Leftrightarrow x_2 \wedge x_3 \tag{5.2}$$

Bei einem Neustart (siehe 5.3.4)des CDCL-Algorithmus wird die gesamte Heuristik neuberechnet.

5.3.2 Unitpropagation

Die Unitpropagation, oder auch BCP, ist einer der wichtigsten Algorithmen für moderne SAT-Solver. In der Unitpropagation werden Klauseln, die nur ein unbelegtes Literal besitzen und noch nicht als SAT evaluiert sind, so belegt, dass Sie wahr werden (siehe 4.1).

Listing 5.3: Funktion für Unitpropagation

```
1  unitPropagation :: ClauseList -> TupleClauseList -> Level ->
       MappedTupleList -> TriTuple
2  unitPropagation clist tlist lvl mapped
3
4      | null clist || null fstElem = (clist, tlist, mapped)
5      | otherwise = unitPropagation resolutionC
6                    (tlist ++ [(calcTuple, ogClause)]) lvl updatedMap
7      where
8            unitClause = getUnitClause clist
9            fstElem = getClauseFromReducedClauseAndOGClause unitClause
10           calcTuple = setVariable fstElem
11           ogClause = Reason (getOGFromReducedClauseAndOGClause
                 unitClause)
12           updatedMap = pushToMappedTupleList mapped lvl calcTuple
                 ogClause
13           subsumptionC = unitSubsumption clist calcTuple
14           resolutionC = unitResolution subsumptionC calcTuple
```

Zunächst wird in 5.3 überprüft, ob die ClauseList Elemente besitzt. Wenn dies der Fall ist, wird nach einer Klausel gesucht, die nur ein unbelegtes Literal (Unit-Clause) besitzt. Wenn jedoch keine Elemente mehr vorhanden sind oder keine Unit-Clause gefunden wird, werden die übergebenen Daten in Zeile 4 zurückgegeben und der Algorithmus wird beendet.
Ansonsten setzt der Algorithmus das Literal so, dass die Klausel wahr wird und wendet Subsumption (unitSubsumption in Zeile 14 im Listing 5.3) auf die ClauseList an. Die neue ClauseList wird dann mit Resolution (unitResosultion in Zeile 15 im Listing 5.3) verändert. Gleichzeitig wird der Grund für die Entscheidung des Algorithmus in die MappedTupleList aufgenommen. Solange der erste Fall nicht eintritt, arbeitet der Algorithmus rekursiv mit den bearbeiteten Daten weiter.
Subsumption ist das Löschen einer ganzen Klausel, die durch die Belegung zu SAT evaluiert wird, während die Resolution das Löschen eines Literals ist, das zu einem logischen Falsch beurteilt wird (siehe Teilregel 2 von ersten Regel).

5.3.3 Konfliktanalyse

Bei der Konfliktanalyse wird die leere Klausel mithilfe der Literalbelegung analysiert und es werden neue Klauseln gelernt, die die Belegung der Entscheidungsliterale zu einem bestimmten Wert erzwingt. Wenn der Konflikt in Level 0 gefunden wurde (siehe Zeile 5 im Listing 5.4), wird das Programm gestoppt. Ansonsten berechnet „analyzeConflict“ mithilfe der Funktion „calcReason“ in Zeile 7 im Listing 5.4 die neue Klausel, wobei 1UIP als Stopkriterium verwendet wird. Dabei wird die Resolution verwendet, um die Klausel zu berechnen. Danach werden die gelernte Klausel,

die aktualisierte MappedTupleList, ActivityMap und das neue Level als Ergebnis zurückgegeben.

Listing 5.4: Funktion für Konfliktanalyse

```
1 analyzeConflict :: Level -> Clause -> MappedTupleList -> ActivityMap ->
      (Level, Clause, MappedTupleList, ActivityMap)
2 analyzeConflict lvl emptyClause mtl aMap
3
4     -- Case: Given Level is 0. Return -1
5     | getLevel lvl == 0 = (Level (-1), [], mtl, aMap)
6     | otherwise = (decreaseLvl lvl, fst newCl, updatedMtl, snd newCl)
7     where reason = calcReason lvl emptyClause mtl
8           updatedMtl = deleteLvl lvl mtl
9           newCl = addClause reason aMap
```

5.3.4 Neustarts

Neustarts werden in CDCL SAT-Solver verwendet, um die Suche in Teilbäumen, die zu keiner Lösung führen können, zu unterbrechnen [8]. Dabei werden alle aktuellen Belegungen und Heuristiken gelöscht. Die gelernten Klauseln bleiben erhalten und die Heuristiken werden neu berechnet. Mit diesen Daten fängt der Algorithmus dann mit dem CDCL Prozess wieder an.
Für die Zeitplanung der Neustarts wird eine Sequenz verwendet, die auf der Luby-Sequenz basiert. Die Luby-Sequenz wurde von Michael Luby, Alistair Sinclair und David Zuckermann 1993 vorgestellt [16]. Sie wurde als Strategie für eine Minimierung der Laufzeit entwickelt, in der eine Antwort für ein Las Vegas-Algorithmus gefunden wird. Dabei sind Las Vegas-Algorithmen Algorithmen, die für beliebige Inputs ein korrektes Ergebnis finden. Die Laufzeit dieser Algorithmen ist jedoch unterschiedlich.
Die verwendete Sequenz für Neustarts sieht folgendermaßen aus:

$$n,\ n*2,\ n,\ n*2,\ n*4,\ n,\ n*2,\ n*4,\ n*8, \ldots \tag{5.3}$$

Falls das Programm in einen Konflikt gerät, so wird ein Integer (conflictIteration) hochgezählt. Wenn conflictIteration den gesetzten Wert für eine Grenze erreicht, so wird der CDCL-Algorithmus neugestartet, wobei conflictIteration wieder auf 0 gesetzt wird. In der Implementierung wurde für $n = 20$ ausgewählt.

6 Auswertung der Implementierung

In diesem Kapitel wird die Korrektheit der Implementierung des SAT-Solvers mit dem PicoSAT-Solver verglichen. Dabei wird erklärt, wie dieser Vergleich im Programm funktioniert. Nach dem Vergleich der Korrektheit wird die Performance der Implementierung mit den Solvern zChaff und MiniSAT verglichen.

6.1 Vergleich der Korrektheit mit PicoSAT

Um auf die Korrektheit der Implementierung zu prüfen, werden die Ergebnisse des SAT-Solver gegen den PicoSAT-Solver geprüft. Der PicoSAT-Solver wurde hierfür ausgewählt, weil dieser eine einfache Anbindung zu Haskell besitzt. Zur Generierung der Tests wird das Haskellpaket QuickCheck verwendet.

Listing 6.1: Code zum Vergleich der Richtigkeit der Implementierung

```
1  prop_picoSATcomparison :: [[NonZero Int]] -> Property
2  prop_picoSATcomparison cl = withMaxSuccess 10000 ( monadicIO ( do
3    let clauses = coerce cl
4    picoSol <- run (PicoSAT.solve clauses)
5    let cdclSol = cdcl (map (map fromIntegral) clauses) False False
6    assert ( case (picoSol, cdclSol) of
7               (PicoSAT.Unsatisfiable, UNSAT) -> True
8               (PicoSAT.Unknown, _)           -> False
9               (PicoSAT.Solution _, SAT _)    -> True
10              _                              -> False )))
```

Die Funktion erstellt zufällig 10000 verschiedene Probleme, wobei die SAT-Solver die generierten Probleme auf ihre Ergebnisse vergleichsweise prüfen. Dabei werden die SAT-Instanzen auf vier verschiedene Fälle verglichen. Wenn beide Solver entsprechende Ergebnisse mit UNSAT für ein Problem melden, so ist der Test für diese eine Instanz erfolgreich. Dies ist auch der Fall, wenn PicoSAT Solution [Integer] und die Implementierung SAT TupleList als Ergebnis ausrechnen. Falls PicoSAT ein Unknown Ergebnis kalkuliert oder keine der beiden ersten genannten Fälle in einem

SAT-Problem ausgerechnet werden, so wird der Test als fehlgeschlagen ausgewertet. Wenn ein einziger Test dabei fehlschlägt, so wird der ganze Test abgebrochen. Der fehlgeschlagene Test, bei dem ein unterschiedliches Ergebnis berechnet wurde, wird dann in der Konsole angezeigt
Mit der Eingabe des Befehls „stack test“ in der Kommando-Zeile innerhalb des Implementierungsordners wird der Test gestartet. Die derzeitige Version der Implementierung kann die generierten Test ohne Probleme ausführen, womit die Richtigkeit des Solvers mit großer Wahrscheinlichkeit gegeben ist.

6.2 Performancevergleich mit anderen SAT-Solvern

Der Performancevergleich der Solver wird innerhalb einer virtuellen Maschine (VM) durchgeführt. Die Spezifikation für die VM hierbei sind folgende:

- Betriebsystem: Ubuntu (64-bit)
- Speicherort: Hard Disk Drive (HDD)
- Hauptspeicher 2048 MB
- 1 Prozessor

Die VM läuft auf einem Laptop mit den folgenden Spezifikationen:

- Prozessor: Intel(R) Core(TM) i5-8250U CPU @1.60GHz 1.80GHz
- RAM: 8,00 GB
- Systemtyp: 64-Bit-Betriebssystem, x64-basierter Prozessor
- Betriebssystem: Windows 10 Home
- Version: 20H2

In dem Vergleich werden die Benchmark Very Large SAT[1] von „Construction and Analysis of Distributed Processes" (CADP) verwendet. Des Weiteren werden auch einzelne Benchmarks getestet, die auf der SATLIB Webseite[2] publiziert sind [9].
Die Ausführung der Solver wird gestoppt, wenn nach 5 Minuten Echtzeit keine Ergebnisse für die entsprechende Probleminstanz vorliegen. Zum Vergleich werden zChaff 2007 .3.12 (64-bit)[3] und MiniSAT 2.2.1-5build2_amd64[4] herangezogen. Die

[1] Link zur Benchmark: `https://cadp.inria.fr/resources/vlsat/`
[2] Link zur Webseite:`https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`
[3] http://www.princeton.edu/∼chaff/zchaff.html
[4] Installation mithilfe von „apt-get install minisat“

Implementierung wird in dieser Arbeit als „Impl“ abgekürzt. In dem Vergleich werden die benötigte Zeit zum Lösen der Probleme, die Anzahl der Neustarts, Entscheidungen und der gelernten Klauseln geprüft. Für die Messung der Laufzeit wird in Impl das Haskellpaket timeit 2.0[5] verwendet und alle Zeiten werden 2 Stellen nach dem Komma gerundet. zChaff gibt keine Auskunft zur Anzahl der Neustarts. Für die Statistik von MiniSAT wird angenohmen, dass die Anzahl der Konflikte gleich der Anzahl der gelernten Klauseln ist.

vlsat1_10_17.cnf

	Impl	zChaff	MiniSAT
Prozessorzeit	0.00	0.00	0.00
Anzahl der Neustarts	0	-	1
Entscheidungen	0	1	1
Anzahl der gelernten Klauseln	0	0	0

Tabelle 6.1: VLSAT, die Benchmark hat 10 verschiedene Literale und 17 Klauseln

vlsat1_54_270.cnf

	Impl	zChaff	MiniSAT
Prozessorzeit	0.01	0.00	0.00
Anzahl der Neustarts	0	-	1
Entscheidungen	1	1	3
Anzahl der gelernten Klauseln	0	0	0

Tabelle 6.2: VLSAT, die Benchmark hat 54 verschiedene Literale und 270 Klauseln

vlsat1_210_1275.cnf

	Impl	zChaff	MiniSAT
Prozessorzeit	7.42	0.00	0.00
Anzahl der Neustarts	1	-	1
Entscheidungen	8	33	1
Anzahl der gelernten Klauseln	59	0	0

Tabelle 6.3: VLSAT, die Benchmark hat 210 verschiedene Literale und 1275 Klauseln

[5] `https://hackage.haskell.org/package/timeit`

vlsat1_222_1477.cnf

	Impl	zChaff	MiniSAT
Prozessorzeit	50.53	0.00	0.00
Anzahl der Neustarts	7	-	1
Entscheidungen	6	20	20
Anzahl der gelernten Klauseln	321	2	1

Tabelle 6.4: VLSAT, die Benchmark hat 222 verschiedene Literale und 1477 Klauseln

vlsat1_228_3437.cnf

	Impl	zChaff	MiniSAT
Prozessorzeit	2.53	0.00	0.00
Anzahl der Neustarts	0	-	1
Entscheidungen	2	5	4
Anzahl der gelernten Klauseln	8	0	0

Tabelle 6.5: VLSAT, die Benchmark hat 228 verschiedene Literale und 3437 Klauseln

flat100-1.cnf

	Impl	zChaff	MiniSAT
Prozessorzeit	84.30	0.00	0.00
Anzahl der Neustarts	12	-	1
Entscheidungen	18	63	52
Anzahl der gelernten Klauseln	645	22	30

Tabelle 6.6: SATLIB, Färbung eines Graphen mit 100 Knoten und 239 Kanten. Das Problem hat 300 verschiedene Literale und 1117 Klauseln.

flat100-100.cnf

	Impl	zChaff	MiniSAT
Prozessorzeit	9.76	0.00	0.00
Anzahl der Neustarts	3	-	1
Entscheidungen	16	71	34
Anzahl der gelernten Klauseln	99	33	6

Tabelle 6.7: SATLIB, Färbung eines Graphen mit 100 Knoten und 239 Kanten. Das Problem hat 300 verschiedene Literale und 1117 Klauseln.

flat125-70.cnf

	Impl	zChaff	MiniSAT
Prozessorzeit	-	0.00	0.00
Anzahl der Neustarts	-	-	1
Entscheidungen	-	62	58
Anzahl der gelernten Klauseln	-	13	37

Tabelle 6.8: SATLIB, Färbung eines Graphen mit 125 Knoten und 301 Kanten. Das Problem hat 375 verschiedene Literale und 1403 Klauseln. Die Implementierung hat keine Lösung innerhalb der Vorgaben für die Echtzeit gefunden.

Es wird ersichtlich, dass die Implementierung in der Performance schlechter abschneidet als etablierte SAT-Solver. Hierbei gibt es mehrere Gründe, die diesen Unterschied in der Performance erklären können.
Zum einem ist die Verwendung einer Datenstruktur, die keine Watched Literals unterstützt, ein Grund. Während bei der Watched Literals über die Referenzen geprüft werden kann, ob eine Klausel SAT ist, ist dies nicht der Fall bei der Implementierung. Dies trägt dazu bei, dass die Implementierung bei dieser Überprüfung länger braucht. Ein weiteres Problem der Datenstruktur ist die ständige Permutation der Daten nach einer Entscheidung und dem BCP-Prozess. Durch die Betrachtung eines Profilings mit der Benchmark vlsat1_222_1477.cnf[6] wird ersichtlich, dass der Großteil der Zeit in *calculateClauseList* verbracht wurde. Insgesamt verbringt das Programm 79,4% seiner Zeit in *calculateClauseList*. *calculateClauseList* ist eine Hilfsfunktion, die Subsumption und Resolution nach einer Entscheidung anwendet, um die Klauseln in der Formel zu verändern.
Ein weiteres Problem ist die Implementierung des Entscheidungsalgorithmus und der Konfliktanalyse. Beim Austesten des Solvers mit einem Backjumping-Algorithmus wurde offentsichtlich, das kein Backjumping in der Implementierung auftrat. Die Konflikte entstehen bei dem Solver immer nur dann, wenn das zweithöchste Level gleich dem höchsten Level - 1 ist. Deshalb ist der Solver immer nur um ein Level zurückgegangen. Daraus wird ersichtlich, dass die Implementierung nicht vom Prinzip des Backjumpings profitiert.

[6] siehe `https://hmpham97.github.io/BA-pages/`

7 Fazit

Das Ziel dieser Bachelorarbeit war es, eine rein funktionelle Implementierung eines CDCL SAT-Solvers zu erstellen. Dafür wurden Literaturarbeiten betrachtet, die mit Haskell, SAT-Problemen und CDCL in Verbindung stehen.

Die Ergebnisse der Arbeit zeigen, dass eine Entwicklung eines funktionierenden CDCL SAT-Solvers in Haskell möglich ist. Die meisten Konzepte, die für einen CDCL SAT-Solver notwendig sind, konnten in der Arbeit umgesetzt werden. Beim Vergleich der Korrektheit wird ersichtlich, dass der Solver generierte Tests in kurzer Zeit lösen kann. Daraus kann geschlossen werden, dass kleine SAT-Probleme dem Solver keine Probleme darstellen und dass die Richtigkeit der Implementierung mit großer Wahrscheinlichkeit gegeben ist.

Jedoch wurde im Performancevergleich mit etablierten SAT-Solvern offensichtlich, dass die Implementierung noch viele Verbesserungen benötigt, damit sie mit Solvern wie zChaff und MiniSAT konkurrieren kann. Große Diskrepanzen konnten beim Lernen von Klauseln und der benötigten Zeit zum Lösen größerer SAT-Probleme beobachtet werden. Um diese Probleme lösen zu können, werden Änderungen in der Datenstruktur, im Entscheidungsalgorithmus und in der Konfliktanalyse notwendig.

Durch die Literaturarbeit wurde somit gezeigt, dass eine rein funktionelle Implementierung eines CDCL SAT-Solvers möglich ist. Um jedoch eine performanten Solver zu erstellen, werden Änderungen in der Programmierung benötigt. Es wird ersichtlich, wenn diese Änderungen implementiert sind, dass eine rein funktionale Implementierung von CDCL SAT-Solvern sinnvoll sein kann.

Dokumentation für Code

Für eine umfangreichere Dokumentation des Codes wird auf Haddock Dokumentation / auf den Sourcecode hingewiesen. Diese Dokumentation dient nur als grober Überblick für die hauptsächlichen Funktionen, die implementiert wurden. Einfache Getter-Funktionen werden hier nicht dokumentiert.

Types.hs

Die Datei enthält alle nötigen Datentypen, die für die Implementierung des CDCL Programms notwendig sind.
data
CDCLResult
Rückgabewert von CDCL Funktion. Folgende Datentypen können hier zurückgegeben werden:

- SAT TupleList
- SAT_WITH_STATS TupleList Integer Integer Integer Integer
- SAT_WITH_FULL_STATS TupleList MappedTupleList [Clause] Integer Integer Integer Integer
- UNSAT
- UNSAT_WITH_STATS [Clause] [Clause] (Die erste Liste sind gelernte Klauseln, während die zweite Klauseln sind, die Konflikte hervorgeführt haben)

Reason
Reason zeigt an, aus welchem Grund eine Belegung für eine Variable entschieden wurde. Diese können folgende sein:

- Decision
- Reason Clause

BoolVal
BoolVal gibt den gesetzten Wert einer Variablen zurück. Diese sind BTrue, BFalse or BNothing. Equivalente Werte für diese sind 1, 0 und -1.
InterpretResult
InterpretResult ist das Ergebnis von der Interpret Funktion. Ergebnisse können folgende sein:

- OK
- NOK Clause
- UNRESOLVED

newtype
Literal
Datentyp für Literal. Die Darstellung für diesen Datentyp ist Lit Integer.
Bsp: Lit 1
Level
Datentyp für Level. Die Darstellung für diesen Datentyp ist Level Integer. Es zeigt an auf welcher Entscheidungsebene ein Literal belegt wurde.
Bsp: Level 2
Activity
Datentyp für Activity. Die Darstellung für diesen Datentyp ist Activity Integer. Activity wird für die Aktivitätsheuristik der Literale verwendet.
Bsp: Activity 9
Period
Datentyp für Period. Die Darstellung für diesen Datentyp ist Period Integer. Wenn Period den Wert 0 hat, so wird die aktuelle Aktivitätsheuristik halbiert und Period wird wieder auf ihren Startwert gesetzt.
Bsp: Period 8

Type
Clause
Clause ist eine Liste bestehend aus Literalen. Bsp: [Lit 1, Lit 2]
ReducedClauseAndOGClause
Synonym für Tupel, die aus zwei Clause bestehen. Die erste Clause wird durch Funktionen gekürzt, während die zweite Clause im Tupel in ihrem Originalzustand bleibt. Bsp: ([Lit 1, Lit 2], [Lit 1, Lit 2, Lit 3])

ClauseList
ClauseList ist eine Liste aus ReducedClauseAndOGClause.
Bsp: [([Lit 1, Lit 2], [Lit 1, Lit 2, Lit 3]), ([Lit 1, Lit 2], [Lit 1, Lit 2])
Tuple
Synonym für Tupel, die aus Literal und BoolVal bestehen.
Bsp: (Lit 1, BFalse)
TupleList
Liste aus Tuple. Wird für den Datentyp CDCLResult verwendet.
TupleClause
Synonym für Tupel, die aus Tuple und Reason bestehen.
Bsp: ((Lit 1, BFalse), Decision)
TupleClauseList
Liste aus TupleClause.
MappedTupleList
Eine Map, die Level als Key verwendet und TupleClauseList als Value besitzt.
ActivityMap
Variablen sind in dieser Map Keys, während Activity Values darstellen
LiteralActivity
Synonym für Tupel, die aus Literal und Activity bestehen.
Bsp: (Lit 1, Activity 3)
TriTuple
Eine Tupel aus drei Elementen. Diese enthält ClauseList, TupleClauseList und MappedTupleList. Dieser Typ wird nur im Unitpropagation-Modul verwendet.

Algorithm.hs

cdcl
Funktion benötigt drei Parameter. Der erwartete Parameter ist eine Liste aus Listen, welche mit Integern gefüllt ist. Die beiden anderen Parameter sind Booleans. Entsprechende Boolean Belegungen geben entweder mehr oder weniger Statistiken zurück. Ruft die rekursive cdcl' Funktion auf. Diese cdcl' Funktion führt den Prozess ganzen Prozess eines CDCL SAT-Solvers durch und gibt CDCLResult als Ergebnis zurück.
calculateClauseList
Funktion benötigt ClauseList und TupleClauseList als Parameter. Diese Funktion wird aufgerufen, wenn eine Variable BoolVal durch eine Decision erhält und berech-

net ihr Ergebnis mithilfe von unitSubsumption und unitResolution. Der Rückgabewert ist eine veränderte ClauseList.
interpret
Interpretiert eine gegebene ClauseList mithilfe einer übergebenen TupelClauseList. Dabei wird rekursiv nach der Reihe eine Clause interpretiert, bis alle Klauseln interpretiert sind oder eine Klausel nicht zu OK evaluiert wird.
Rückgabewerte für diese Funktion sind InterpretResult.
searchTupel
Gibt den Wert eines Tupelpaares zurück basiered auf dem gegebenen Variablenwert und der TupelList. Dabei wird ein BoolVal zurückgegeben.

Unitpropagation.hs

unitPropagation
Führt das Unit-Propagation Verfahren durch. Die Funktion erwartet ClauseList, TupelClauseList, Level und MappedTupleList als Argumente. Als Ergebnis wird ein TriTuple zurückgegeben.
getUnitClause
Gibt eine ReducedClauseAndOGClause durch das überprüfen einer ClauseList zurück. Dabei wird überprüft ob das erste Element im ReducedClauseAndOGClause die Länge 1 besitzt.
setVariable
Setz einen Wert von BFalse oder BTrue in einem Tuple und gibt dieses zurück.
unitSubsumption
Löscht Klauseln aus ClauseList, wenn Klauseln gefunden werden, die Variablen enthalten, welche bei einem eingesetzem Tupelwert zu 1 evaluiert werden. Die bearbeitete ClauseList wird dann zurückgegeben.
unitResolution
Löscht Variablen aus den Clauseln, die zu 0 evaluiert werden. Als Ergebnis wird die bearbeitete ClauseList weitergegeben.

Decisionalalgorithm.hs

initialActivity
Die Funktion wird initial bei der Verwendung von cdcl benutzt und erwartet eine ClauseList und ActivityMap als Paremeter. Wenn der CDCL-Prozess neugestartet wird, so wird die Funktion nochmal aufgerufen. Mithilfe von rekursiven Aufrufen

von initialActivity und updateActivity wird dann eine ActivityMap berechnet und zurückgegeben.
updateActivity
UpdateActivity benötigt eine Clause und ActivityMap als Parameter. Durch Rekursion erhalten neue Variablen einen Eintrag in die ActivityMap, während bestehende um eins aktualisiert werden.
halveActivityMap
Die Funktion halbiert alle Activity nach einer bestimmten Period. Als Parameter werden eine ActivityMap und eine Liste von allen Variablen benötigt.
getHighestActivity
Erwartet ClauseList, ActivityMap und eine Liste von VariableActivity als Parameter und gibt eine Liste von VariableActivity mit der höchsten Activity als Ergebnis zurück.
getShortestClauseViaActivity
Beim Aufruf der Funktion werden zwei ClauseList und eine Liste von VariableActivity benötigt. Die Funktion gibt eine ClauseList mit den kürzesten Clause zurück, die die gegebenen VariableActivity enthalten.
checkClauseForLiteral
Bekommt eine Clause und LiteralActivity-Liste als Parameter. Die Funktion überprüft ob ein Literal von der Liste in der Clause vorhanden ist. Gibt entsprechend ein Boolean als Ergebnis zurück.
setLiteralViaActivity
SetLiteralViaActivity benötigt eine Clause und LiteralActivity. Basierend auf das Literal in der Clause wird ein TupleClause als Ergebnis zurückgegeben. Ist das Literal negiert, so wird BTrue gesetzt. Bei einem Literal ohne Vorzeichen wird das Literal mit BFalse belegt.

MapLogic.hs

pushToMappedTupleList
Die Funktion erwartet MappedTupleList, Level, Tuple und ein Reason. Anhand der übergebenen Parameter wird die MappedTupleList aktualisiert und anschließend zurückgegeben.
deleteLvl
Die Funktion löscht das gegebenene Level aus einer gegebenen MappedTupleList und gibt diese zurück.

Conflict.hs

analyzeConflict
Die Funktion erwartet folgende Argumente: Level, Clause, MappedTupleList, ClauseList und ActivityMap. Mithilfe von calcReason wird eine neue Clause berechnet und durch addClause zu der gegebenen ClauseList hinzugefügt. Dabei wird durch deleteLvl das letzte Level aus MappedTupleList gelöscht. Level, ClauseList, MappedTupleList und ActivityMap wird am zurückgegeben.
calcReason
Berechnet mithilfe von Level, Clause und MappedTupleList eine 1UIP-Clause.

CDCLFilereader.hs

readCdclFile
Die Funktion erwartet einen String Input und zwei Booleans und gibt eine IO Ausgabe zurück. Der String Input soll hierbei eine existierende cnf-Datei sein. Bei einem richtigen Input ruft die Funktion loopCheck auf und gibt ihr Ergebnis aus. Beispiel für einen Aufruf in Windows:
CDCL-exe.exe -i test.cnf
loopCheck
Die Funktion überprüft, ob eine Zeile mit einem Integer oder einem - anfängt. Wenn dies der Fall ist, wird die gesamte Zeile in ein Integerliste hinzugefügt. Diese Liste wird dann zu einer Liste von Integerlisten beigefügt. Wenn die Funktion das Ende der Datei erreicht, wird die die Liste von Integerlisten der cdcl-Funktion übergeben. Das Ergebnis wird dann als IO (Maybe CDCLResult) zurückgegeben.

Main.hs

Das Programm kann folgendermaßen gestartet werden.
Zuerst muss ein **stack install** durchgeführt werden.
Windows:

- CDCL-exe.exe -i test.cnf [-f | -s] (Option -f gibt alle Statistiken zurück. -s gibt weniger Daten zurück)

Linux:

- PFAD_zur_exe/CDCL-exe -i test.cnf [-f | -s]

- Oder: hinzufügen der exe zum bin. Dann kann das Programm von überall folgendermaßen gestartet werden:
  - CDCL-exe -i test.cnf [-f | -s]

Bibliography

[1] Stephan Augsten. *Was ist ein Programmierparadigma?* de. URL: https://www.dev-insider.de/was-ist-ein-programmierparadigma-a-864056/ (visited on 06/22/2021).

[2] Armin Biere. "Chapter 18. Bounded Model Checking". en. In: *Frontiers in Artificial Intelligence and Applications*. Ed. by Armin Biere et al. IOS Press, 2009. ISBN: 978-1-64368-160-3 978-1-64368-161-0. DOI: 10.3233/FAIA201002. URL: http://ebooks.iospress.nl/doi/10.3233/FAIA201002 (visited on 07/27/2021).

[3] Armin Biere. *Kissat SAT Solver*. URL: http://fmv.jku.at/kissat/ (visited on 07/03/2021).

[4] Adrienne Bloss, Paul Hudak, and Jonathan Young. "Code optimizations for lazy evaluation". en. In: *Lisp and Symbolic Computation* 1.2 (Sept. 1988), pp. 147–164. ISSN: 0892-4635, 1573-0557. DOI: 10.1007/BF01806169. URL: http://link.springer.com/10.1007/BF01806169 (visited on 07/13/2021).

[5] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *In Stoc*. ACM, 1971, pp. 151–158.

[6] Martin D. Davis, G. Logemann, and D. Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5 (1962), pp. 394–397.

[7] Martin D. Davis and H. Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7 (1960), pp. 201–215.

[8] Niklas Eén and Niklas S"orensson. *An Extensible SAT-solver*. en. 2003. URL: http://minisat.se/downloads/MiniSat.pdf (visited on 07/02/2021).

[9] Holger H Hoos and Thomas Stützle. "SATLIB: An Online Resource for Research on SAT". en. In: (2000), p. 12.

[10] Paul Hudak et al. "A history of Haskell: being lazy with class". en. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. San Diego California: ACM, June 2007, pp. 3–5. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856. URL: https://dl.acm.org/doi/10.1145/1238844.1238856 (visited on 06/30/2021).

[11] Roberto J Bayardo Jr and Robert C Schrag. "Using CSP Look-Back Techniques to Solve Real-World SAT Instances". en. In: (), p. 6.

[12] Henry Kautz and Bart Selman. "Planning as Satisfiability". en. In: (1992), p. 12. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=030430E32867DFAE652F1E0848E797CC?doi=10.1.1.35.9443&rep=rep1&type=pdf (visited on 07/28/2021).

[13] Daniel Kroening and Ofer Strichman. "Decision Procedures for Propositional Logic". In: *Decision Procedures: An Algorithmic Point of View*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 27–58. ISBN: 978-3-662-50497-0. DOI: 10.1007/978-3-662-50497-0_2. URL: https://doi.org/10.1007/978-3-662-50497-0_2.

[14] Christian Lengauer. *Was ist funktionale Programmierung?* URL: https://www.infosun.fim.uni-passau.de/cl/lehre/funcprog05/wasistfp.html (visited on 06/22/2021).

[15] L A Levin. "Universal Sequential Search Problems". ru. In: *Probl. Peredachi Inf.* 9.3 (1973), pp. 115–116.

[16] Michael Luby, Alistair Sinclair, and David Zuckerman. "Optimal Speedup of Las Vegas Algorithms". en. In: (1993), p. 13.

[17] Simon Marlow. *Haskell 2010 Language Report*. July 2021. URL: https://www.haskell.org/definition/haskell2010.pdf (visited on 07/01/2021).

[18] Jo Marques et al. "GRASP—A New Search Algorithm for Satisfiability". In: *in Proceedings of the International Conference on Computer-Aided Design*. 1996, pp. 220–227.

[19] J.P. Marques-Silva and K.A. Sakallah. "GRASP: a search algorithm for propositional satisfiability". en. In: *IEEE Transactions on Computers* 48.5 (May 1999), pp. 506–521. ISSN: 00189340. DOI: 10.1109/12.769433. URL: http://ieeexplore.ieee.org/document/769433/ (visited on 07/16/2021).

[20] Matthew W Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". en. In: (), p. 6. URL: http://www.princeton.edu/~chaff/publication/DAC2001v56.pdf (visited on 02/23/2021).

[21] Stefania Loredana Nita and Marius Mihailescu. "Functional Programming". In: *Haskell Quick Syntax Reference: A Pocket Guide to the Language, APIs, and Library*. Ed. by Stefania Loredana Nita and Marius Mihailescu. Berkeley, CA: Apress, 2019, pp. 1–3. ISBN: 978-1-4842-4507-1. DOI: 10.1007/978-1-4842-4507-1_1. URL: https://doi.org/10.1007/978-1-4842-4507-1_1.

[22] Edward G. Nokie. *Side Effects*. URL: https://www.radford.edu/nokie/classes/320/Tour/side.effects.html (visited on 07/14/2021).

[23] Hrsg: Satisfiability: Application and Theory (SAT) e.V. *SAT Basics*. URL: http://satassociation.org/articles/sat.pdf (visited on 07/05/2021).

[24] Hrsg: Satisfiability: Application and Theory (SAT) e.V. *SAT Competitions*. URL: http://satcompetition.org/ (visited on 07/03/2021).

[25] Carsten Sinz. *SAT-Race 2006*. URL: http://fmv.jku.at/sat-race-2006/results.html (visited on 07/06/2021).