Hochschule für angewandte Wissenschaften
München

Fakultät für Informatik und Mathematik

Bachelorarbeit
zur Erlangung des Grades
Bachelor of Science
im Studiengang Informatik

Entscheidungsprozedur für Bitvektor in Haskell

Autor:	Korbinian Wieser
Matrikelnummer:	46194820
Abgabedatum:	10. Oktober 2024
Aufgabensteller:	Prof. Dr.-Ing. Matthias Güdemann
Betreuer:	Prof. Dr.-Ing. Matthias Güdemann

Hochschule München University of Applied Sciences

Diese Erklärung ist zusammen mit der Bachelorarbeit bei der Prüfer:in abzugeben.

Wieser, Korbinian	Lorenzenberg, 10.10.2024
(Familienname, Vorname)	(Ort, Datum)
14.06.2002	7W / 24 20 25
(Geburtsdatum)	(Studiengruppe / WiSe/SoSe)

Erklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

(Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Mit der Erfindung des Computers begann die bedeutendste technologische Entwicklung des 20. Jahrhunderts, die bis heute andauert. Computersysteme sind heutzutage in allen Bereichen des Lebens aufzufinden und sind aus der Wissenschaft, Wirtschaft und persönlichem Gebrauch nicht mehr wegzudenken. Sie erleichtern unser tägliches Leben erheblich und sind im Stande hochkomplexe Berechnungen innerhalb kürzester Zeit zu lösen.

Im Laufe der Zeit wurden aus unhandlichen, zimmergroßen Computern Hochleistungsrechner, die in jede Hosentasche passen. Doch mit der fortschreitenden Entwicklung der Computersysteme haben sich nicht nur die Einsatzmöglichkeiten dieser Systeme vervielfältigt, sondern auch ihr Umfang ist erheblich gestiegen. Durch die steigende Komplexität rücken bereits bestehende Probleme, wie die Fehleranfälligkeit von Computersystemen, in den Fokus.

Die vorliegende Arbeit beschäftigt sich mit der Fehleranfälligkeit von Computersystemen und untersucht, welche Entscheidungsprozeduren implementiert werden können, um dieser Problematik entgegenzuwirken. [2, S. 149] Hierzu können Entscheidungsprozeduren zur Validierung eingesetzt werden. Allgemein nutzen Computersysteme Bitvektoren als zentralen Bestandteil, um Informationen zu codieren. [2, S. 149] Aufgrund der limitierten Länge von Bitvektoren kommt es dazu, dass sich diese teilweise bei arithmetischen Operationen wider Erwarten unterschiedlich verhalten. [2, S. 149] Da Bitvektoren einen sehr wichtigen Bestandteil in Computern bilden und auch zur Fehleranfälligkeit dieser Systeme beitragen, werden diese im Zusammenhang von Entscheidungsprozeduren im Verlauf der Arbeit genauer betrachtet. Die Theorie hinter Entscheidungsprozeduren mit Bitvektoren existiert bereits, wobei sich die Frage stellt, wie die Theorie hinter Entscheidungsprozeduren mit Bitvektoren in einer Programmiersprache wie Haskell praktisch umgesetzt werden kann.

Ziel ist es:

1. Bitvektorformeln vollständig in Haskell abbilden zu können,
2. diese mithilfe Bitflattenings in eine aussagenlogische Formel zu verwandeln,
3. anschließend in Tseitin-Form umzuwandeln und
4. mithilfe von SAT-Solving zu lösen.

Abschließend soll das Ergebnis auf Korrektheit geprüft und die Bitvektorformeln programmatisch in Haskell validiert werden. Dank immer effizienter arbeitenden SAT-Solvern für Erfüllbarkeitsprobleme der Aussagenlogik (SAT) können noch komplexere SAT-Probleme, die beispielsweise beim Bit-Blasting entstehen, gelöst werden. [11, S. 2]

Hierzu wird vorab die grundlegende Theorie hinter Entscheidungsprozeduren mit Bitvektoren, sowie SAT-Solving erklärt und auf mögliche Erweiterungen der Theorie eingegangen. Schlussendlich soll die praktische Umsetzung in Haskell beschrieben werden. Als Endergebnis sollen in der Praxis Entscheidungsprozeduren mit Bitvektoren in Haskell umgesetzt werden können.

2 Bitvektor

In Bitvektoren lassen sich Informationen jeder Art abspeichern. Die Umwandlung einer Information in einen Bitvektor wird durch Codieren bewerkstelligt. [2, S. 153 f.] Ebenso kann ein Bitvektor später wieder in die originale Information zurückverwandelt werden. [2, S. 153 f.] Die am häufigsten verwendeten Größen von Bitvektoren sind 8, 16, 32 und 64 Bit. Alle heutigen Computersysteme basieren auf der Verwendung von Bitvektoren als Bytes zur Speicherung von Daten. [7]

2.1 Definition

Zunächst sind Bitvektoren ein zentraler Bestandteil der Arbeit und müssen demnach erstmalig definiert werden. Als Bitvektor kann auch eine Bitkette, ein Bitstring oder auch ein Bitarray gemeint sein. [10] Dabei handelt es sich um eine Folge an Bits, wobei ein Bit entweder 1 oder 0 als Wert annehmen kann. [10] Häufig wird der Wert eines Bits auch mit wahr oder falsch beschrieben. Die Abbildung 2.1 zeigt den Aufbau eines Bitvektors auf. Im Folgenden wird eine Folge an Bits mit dem Begriff **Bitvektor** referenziert.

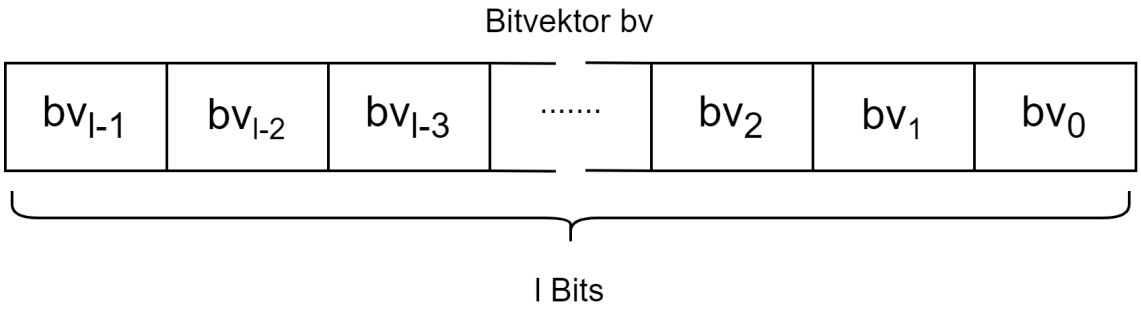

Abbildung 2.1: *Bitvektor der Länge l*

2.2 λ-Notation

Zur Definition von Bitvektoren kann Church's λ-Notation angewandt werden. Dabei legt die Funktion *f(i)* den Wert für jedes i-te Bit fest. [2, S. 151]

$$\lambda i \in \{0,...,l-1\}.f(i) \tag{2.1}$$

λ-Notationsbeispiel:

$$\lambda i \in \{0,...,3\}.1 = 1111 \tag{2.2}$$

2.3 Codierung

Die Codierung von Bitvektoren bezieht sich standardmäßig auf Zahlen. Sie kann sich aber auch auf Texte, Felder etc. beziehen. Bei Betrachtung der Bitvektoroperationen fällt auf, dass es ausschließlich bitweise und arithmetische Operatoren gibt.[2, S. 153] Zunächst kann die Codierung bei den bitweisen vernachlässigt werden. Doch bei den arithmetischen wird sie bei $<$, $>$, $\leq$, $\geq$, $\gg$, $*$ und $\div$ Operatoren benötigt, da diese bei Binärcodierung und Zweierkomplement unterschiedlich gehandhabt werden müssen. [2, S. 154]

Umwandlung von codierten Bitvektoren in ganze Zahlen:
Binärcodierung:

$$\begin{aligned} \langle bv \rangle_u : bv_l &\rightarrow \{0, ..., 2^l - 1\} \\ \langle bv \rangle_u &\doteq \sum_{i=0}^{l-1} bv_i \cdot 2^i \end{aligned} \tag{2.3}$$

Zweierkomplement:

$$\begin{aligned} \langle bv \rangle_s : bv_l &\rightarrow \{-2^{l-1}, ..., 2^{l-1} - 1\} \\ \langle bv \rangle_s &\doteq -2^{l-1} \cdot bv_{l-1} + \sum_{i=0}^{l-2} bv_i \cdot 2^i \end{aligned} \tag{2.4}$$

Codierungsbeispiele:

Binärcodierung:	**Zweierkomplement:**
$\langle 1010 \rangle_u = 10$	$\langle 1010 \rangle_s = -6$

Andere Codierungen von Bitvektoren können ebenso in Entscheidungsprozeduren verwendet werden, wobei im späteren Kapitel 6.2 dabei explizit auf die Anwendungsmöglichkeit der Codierung von Gleitkomma- / Festkommazahlen eingegangen wird.

3 Aufbau Bitvektorformeln

Bei Bitvektorformeln handelt es sich um Ausdrücke der Bitvektorarithmetik. Diese beinhalten bitweise, sowie arithmetische Operatoren für Bitvektoren. [2, S. 153]

3.1 Definition

Der Begriff Bitvektorformel bezieht sich auf eine logische Formel aus Bitvektoren als Variablen und Operatoren, die der Bitvektorarithmetik und der booleschen Logik entstammen. [2, S. 152 f.]

3.2 Bitvektorarithmetik

Zur genaueren Darstellung eines Ausdrucks der Bitvektorarithmetik wird den Operatoren und Operanden Indizes in eckigen Klammern hinzugefügt. [2, S. 153] Diese beschreiben die dort jeweils angewendete Bitbreite / Bitlänge. [2, S. 153] Am Besten kann dies anhand eines Anwendungsbeispieles der Konkatenation zum besseren Verständnis angewendet werden. Hierbei wird ein Vektor a der Länge 7 und b der Länge 13 konkateniert. Das Ergebnis ist insgesamt 20 Bit lang (7 + 13 = 20), die am Operator mitangegeben wird.

Beispiel: Ausdruck einer Konkatenation:

$$a_{[7]} \circ_{[20]} b_{[13]} \tag{3.1}$$

konkretes Beispiel: Ausdruck einer Konkatenation:

$$101 \circ 0110 = 1010110 \tag{3.2}$$

3.2.1 Bitweise Operatoren

Bei den bitweisen Operatoren werden zwei Bitvektoren gleicher Länge mit beliebiger Kodierung gewählt, die als Ergebnis einen Bitvektor gleicher Länge ergeben. [2, S. 153] An den bitweisen Operatoren wird pro jeweiligem Bit der jeweilige Operator angewandt. Zu den bitweisen Operatoren gehören & (AND), | (OR), $\oplus$ (XOR) und $\sim$ (Negation). [2, S. 149]

Beispiel: Bitvektorformel mit &:

$$a_{[8]} \& b_{[8]} = c_{[8]} \tag{3.3}$$

Beispiel: λ-Notation mit &:

$$a \& b \doteq \lambda i.(a_i \wedge b_i) \tag{3.4}$$

3.2.2 Arithmetische Operatoren

Die Verwendung der Arithmetik von Bitvektoren führt oftmals zu einem arithmetischen Überlauf, da das Ergebnis der Länge l $l + x$ Bits zur korrekten Darstellung benötigt, aber nur l Bits vorhanden sind. [2, S. 154] Dementsprechend muss das Ergebnis gekürzt werden. Diese Kürzung wird mithilfe einer Modulo Operation des Ergebnisses mit 2^l erzielt, da das Ergebnis maximal nur 2^l unterschiedlichen Zahlen (z.B. $[0, 2^l - 1]$ bei Binärcodierung) darstellen kann. [2, S. 154] Nachfolgend werden verschiedene arithmetische Operatoren vorgestellt. Die verwendeten Beispiele zu Ausdrücken der Bitvektorarithmetik mit arithmetischen Operatoren basieren auf „Decision Procedures“. [2, S. 155 f.]

- **Addition / Subtraktion**

$$a_{[l]}(+/-)_{[u/s]}b_{[l]} = c_{[l]} \iff \langle a\rangle(+/-)\langle b\rangle = \langle c\rangle \bmod 2^l \tag{3.5}$$

$$a_{[l]u}(+/-)_{[u]}b_{[l]s} = c_{[l]u} \iff \langle a\rangle_u(+/-)\langle b\rangle_s = \langle c\rangle_u \bmod 2^l \tag{3.6}$$

  Die Codierung kann bei der Addition und Subtraktion vernachlässigt werden, da $a(+/-)_u b = a(+/-)_s b$ gilt. [2, S. 155] Durch die Verwendung von einem Volladdierer, der vorzeichenlose und vorzeichenbehaftete Addition und Subtraktion gleichermaßen durchführt, ist das Ergebnis als Bitvektor gesehen gleich. Durch die Codierung des Ergebnisses wird dieses unterschiedlich ausgewertet. Ein Beispiel für eine mögliche Addition bzw. Subtraktion mit gemischter Codierung wurde in Gleichung 3.6 gezeigt.

  konkretes Beispiel: Ausdruck einer Addition mit Überlauf

$$\begin{aligned} 1110 + 0111 = 0101 \iff & 14 + 7 = 21 \bmod 16 \\ & 14 + 7 = 5 \end{aligned} \tag{3.7}$$

- **Multiplikation / Division**

$$a_{[l]}(*/\div)_{[u/s]}b_{[l]} = c_{[l]} \iff \langle a\rangle(*/\div)\langle b\rangle = \langle c\rangle \bmod 2^l \tag{3.8}$$

- **Relationale Operatoren**

$$a_{[l]u} < b_{[l]u} \iff \langle a\rangle_u < \langle b\rangle_u \tag{3.9}$$

$$a_{[l]s} < b_{[l]s} \iff \langle a\rangle_s < \langle b\rangle_s \tag{3.10}$$

$$a_{[l]u} < b_{[l]s} \iff \langle a\rangle_u < \langle b\rangle_s \tag{3.11}$$

$$a_{[l]s} < b_{[l]u} \iff \langle a\rangle_s < \langle b\rangle_u \tag{3.12}$$

  Andere Vergleichsoperatoren verhalten sich identisch zu den oberen Ausdrücken. [2, S. 155] Jedoch verhält sich der Vergleich zwischen zwei unterschiedlich codierten Operatoren unter Verwendung von ANSI-C Compilern anders als oben definiert. [2, S. 155] Dabei wird der vorzeichenbehaftete Operand zu einem vorzeichenlosen

Operanden konvertiert und entspricht nicht dem erwarteten Verhalten. [2, S. 155]

- **Verschiebung**

$$a_{[l]} \ll b_u = \lambda \in \{0,...,l-1\}. \begin{cases} a_{i-\langle b \rangle_u} & : i \geq \langle b \rangle \\ 0 & : \text{sonst} \end{cases} \tag{3.13}$$

Die Verschiebeoperatoren haben zwei Operanden und verschieben den ersten Operanden in die vom Operator (rechts / links) festgelegte Richtung um den decodierten Wert des zweiten Operanden. [2, S. 156]. Bei einer Verschiebung nach links wird der Bitvektor mit Nullen von rechts aufgefüllt. [2, S. 156]

$$a_{[l]u} \gg b_u = \lambda \in \{0,...,l-1\}. \begin{cases} a_{i+\langle b \rangle_u} & : i < l - \langle b \rangle \\ 0 & : \text{sonst} \end{cases} \tag{3.14}$$

$$a_{[l]s} \gg b_u = \lambda \in \{0,...,l-1\}. \begin{cases} a_{i+\langle b \rangle_u} & : i < l - \langle b \rangle \\ a_{l-1} & : \text{sonst} \end{cases} \tag{3.15}$$

Im Gegensatz dazu muss bei der Rechtsverschiebung die Codierung berücksichtigt werden. Auf einem ersten Operanden mit Binärcodierung wird eine logische Rechtsverschiebung angewendet. Dabei wird der Bitvektor von links mit Nullen aufgefüllt. Dagegen wird bei einem ersten Operanden mit Zweierkomplement Codierung eine arithmetische Rechtsverschiebung ausgeführt. Diese füllt den Bitvektor von links mit dem Vorzeichen Bit (a_{l-1}) auf.

- **Erweiterung**

$$extension_{[m]u}(a_{[l]}) = b_{[m]u} \iff \langle a \rangle_u = \langle b \rangle_u \tag{3.16}$$

$$extension_{[m]s}(a_{[l]}) = b_{[m]s} \iff \langle a \rangle_s = \langle b \rangle_s \tag{3.17}$$

Hierbei wird einem Bitvektor m - l Bits vorne angefügt, wobei es sich bei vorzeichenlosen Bitvektoren um eine Nullerweiterung und bei vorzeichenbehafteten Bitvektoren um eine Vorzeichenerweiterung handelt. [2, S. 155] Der daraus entstehende Bitvektor muss mehr Bits als ursprünglich besitzen und darf den darin codierten Wert nicht verändern. [2, S. 155]

3.3 Grammatik

Die verwendete Grammatik für Bitvektorformeln basiert auf dem Kapitel „Bit Vectors“ im Buch „Decision Procedures“. [2, S. 149]
formula : formula ∧ formula | ¬ formula | atom
atom : term rel term | True | False | term[constant]
rel : < | > | ≤ | ≥ | = | ≠
term : term op term | ~ term | identifier | constant | atom?term:term | term[constant:constant]

| extension(term)
op : + | - | ∗ | ÷ | ≪ | ≫ | & | | | ⊕ | ∘
Jede Formel beginnt als *formula*, die zu einer Konkatenation von Formeln, Negierung einer Formel oder einem Atom abgeleitet werden kann. Ein Atom kann zu einer Relation zweier Terme, einem Wahrheitswert oder zu einem *term[constant]* abgeleitet werden. Dabei nimmt *term[constant]* das Bit $bvTerm_{\langle constant \rangle}$ des resultierenden Bitvektors $bvTerm$ (von *term*) als Wahrheitswert an. Zum Hauptbestandteil jeder Bitvektorformel gehört der Term, da dieser die Bitvektoroperationen beinhaltet. Den Term kann zu einer Bitvektoroperation ableitet werden. Darüber hinaus kann ein Term mit ∼ zu einer bitweisen Negierung eines Terms abgeleitet werden. [2, S. 149] Andernfalls gibt es die Möglichkeit den Term zu einem Ausdruck *atom?term:term* mit ternärem Operator abzuleiten. [2, S. 149] Ein ternärer Operator funktioniert identisch zu einem if-else Fall. Zunächst wird das Atom ausgewertet, sodass bei „wahr“ der erste Term und bei „falsch“ der zweite Term als Ergebnis eines Terms verwendet wird. [2, S. 149] Bei Auflösung zu *term[constant:constant]* wird ein Untervektor vom resultierenden Bitvektor eines Terms benutzt. Hingegen wird bei der Ableitung zu *extension(term)* ein Term erweitert. [2, S. 149] Final kann ein Term zu einem Identifikator oder einer Konstante aufgelöst werden. Alle Ausdrücke, die von *formula* aus abgeleitet werden können sind syntaktisch valide Bitvektorformeln.

Ausführendes Beispiel:

$$(1100 \oplus_4 1010) +_4 0110 =_4 result \tag{3.18}$$

Ableitung der Grammatik am ausführenden Beispiel:
start -> formula -> atom -> term rel term -> term op term = result -> (term op term) + 0110 = result -> (1100 ⊕ 1010) + 0110 = result

4 Flattening Bitvektorformeln

4.1 Definition

Als Flattening oder umgangssprachlich „Bit-Blasting“ wird die am Häufigsten verwendete Entscheidungsprozedur für Bitvektorarithmetik bezeichnet. [2, S. 156] Diese wandelt abstrakte Bitvektorformeln in konkrete aussagenlogische Formeln um, die identisch erfüllbar sind. [2, S. 156]

4.2 Vorgehen

Zuerst werden alle Atome der Bitvektorformel mit neuen booleschen Variablen ausgetauscht. [2, S.156] Die daraus entstehende Formel wird das aussagenlogische Skelett der Bitvektorformel genannt. [2, S.156] Dagegen wird die austauschende Variable $e(a)$ zu einem Atom als aussagenlogischer Codierer des Atoms bezeichnet. [2, S.156]

Danach wird allen Termen der Bitvektorformel ein Vektor $e(t)$ derselben Länge, bestehend aus neuen booleschen Variablen, zugewiesen. [2, S.156] Schließlich werden die aussagenlogischen Bedingungen für alle booleschen Variablen der Terme und Atome berechnet, um diese als Konjunktion dem aussagenlogischen Skelett anzufügen. [2, S.157] Des Weiteren hängt die benötigte Bedingung für ein bestimmtes Atom oder einen bestimmten Term individuell vom Inhalt des Atoms oder Terms ab. [2, S. 157] Im Falle eines Bitvektors oder einer booleschen Variable wird keine Bedingung benötigt, weshalb die Bedingung hierbei ersatzweise nur aus einem wahren Wahrheitswert besteht. [2, S. 157] Folgende Bedingung wird verwendet, falls es sich bei Term t um eine Bitvektor Konstante $bv_{[l]}$ handelt. Infolgedessen ist $e(t)_i$ das i-te Bit des Vektors der neuen Variablen von t. [2, S. 156]

$$\bigwedge_{i=0}^{l-1} (bv_i \iff e(t)_i) \tag{4.1}$$

Andererseits muss t eine Bitvektoroperation beinhalten. Die dafür benötigten Bedingungen hängen vom Operator ab. Deren Konstruktion wird in den folgenden Sektionen 4.3 und 4.4 beschrieben.

4.3 Flattening bitweiser Operatoren

Die generierten Bedingungen für die bitweisen Operatoren sind einfach zu erstellen und weisen Ähnlichkeit zu den entsprechenden λ-Notationen auf. [2, S. 157] Demnach wird $e(t)_i$ dem i-ten Bit der λ-Notationsfunktion aus 2.2 gleichgesetzt. Wenn beispielsweise der bitweise Operator & (AND) in $t = a\&_{[l]}b$ verwendet wird, wird die nun folgende

Bedingung erhalten. [2, S. 157]

$$\bigwedge_{i=0}^{l-1}((a_i \wedge b_i) \iff e(t)_i) \tag{4.2}$$

All die anderen bitweisen Operatoren folgen demselben Muster. [2, S. 157] Dafür muss der linken Teil $(a_i \wedge b_i)$ dem gewünschten Operator entsprechend verändert werden.

4.4 Flattening arithmetischer Operatoren

Oftmals folgen die Bedingungen der arithmetischen Operatoren den Implementierungen dieser Operatoren als Schaltkreis. [2, S. 157] Zudem gibt es zahlreiche Literatur zum Bau effizienter Schaltkreise für verschiedene arithmetische Operatoren. [2, S. 158] Infolgedessen wurde in Experimenten gezeigt, dass die einfachsten Schaltkreise normalerweise den SAT-Solver am wenigsten belasten. [2, S. 158]

- **Addition / Subtraktion**
  Zum Flattening wird ein 1 Bit Addierer, der auch Volladdierer genannt wird, verwendet. [2, S. 158] Dieser wird mit den zwei Funktionen *sum* und *carry*, die beide jeweils Bits a, b und cin als Eingabeparameter verwenden, folgendermaßen definiert. [2, S. 158] Dabei berechnet die *carry* Funktion das *carry-out* Bit des Addierers und die *sum* Funktion das Bit des Ergebnisses der Summe. [2, S. 158]

  Definition Volladdierer

$$sum(a,b,cin) \doteq (a \oplus b) \oplus cin \tag{4.3}$$
$$carry(a,b,cin) \doteq (a \wedge b) \vee ((a \oplus b) \wedge cin) \tag{4.4}$$

  Im Folgenden wird die Definition eines 1-bit Addierers auf einen l-bit Addierer mit Bitvektoren beliebiger Länge erweitert. [2, S. 158] Dazu muss zuerst die *sum* und *carry* Funktionen auf l-bit Bitvektoren angewendet werden. Zunächst stellen x und y zwei l-bit Bitvektoren dar. [2, S. 158] cin hingegen ist nur ein einzelnes Bit, das als Übertragsbit für das erste Bit c_0 bei der Berechnung des Ergebnisses und der weiteren Übertragsbits benötigt wird. [2, S. 158]

  Definition Übertragbits

$$c_i \doteq \begin{cases} cin & : i = 0 \\ carry(x_{i-1}, y_{i-1}, c_{i-1}) & : \text{sonst} \end{cases} \tag{4.5}$$

  Ein l-bit Addierer nimmt zwei l-bit Bitvektoren, sowie das cin Bit und erhält daraus das Ergebnis der Summe beider Bitvektoren sowie $cout$. [2, S.158] Die folgende Definition verwendet die eben definierten Übertragbits c_i. [2, S. 158]

Definition Addierer

$$add(x,y,cin) \doteq \langle result, cout \rangle \tag{4.6}$$
$$result_i \doteq sum(x_i, y_i, c_i) \text{ for i} \in \{0,...,l-1\} \tag{4.7}$$
$$cout \doteq c_l \tag{4.8}$$

Die äquivalente Schaltung zum Volladdierer wird Carry-Ripple-Addierer genannt. [2, S.158] Um die Bedingung zu einer Addition wie $t = a + b$ zu implementieren, muss der Addierer mit $cin = 0$ verwendet werden. [2, S. 158]

$$\bigwedge_{i=0}^{l-1} (add(a,b,0).result_i \iff e(t)_i) \tag{4.9}$$

Die Implementierung einer Subtraktion wie $t = a - b$ nutzt die selbe grundlegende Schaltung. [2, S. 158] Anzumerken ist, dass hier mit $\sim b$ bitweise negiert wird und für $cin = 1$ gilt. Diese Änderungen bewirken aufgrund von $\langle(\sim b)+1\rangle_s = -\langle b\rangle_s \bmod 2^l$, dass b negiert wird und dementsprechend als Subtraktion agiert. [2, S. 158]

$$\bigwedge_{i=0}^{l-1} (add(a,\sim b,1).result_i \iff e(t)_i) \tag{4.10}$$

Beweis zu $\langle(\sim b)+1\rangle_s = -\langle b\rangle_s$ mod 2^l:

1. **Bitweise Negation:**
   Die bitweise Negation $\sim b$ flippt alle Bits von b. Ein Bit zu Flippen bedeutet, dass es seinen Wert ändert (0 -> 1 & 1 -> 0) und ist identisch zur bitweisen Negation. Das ist in der Arithmetik des Zweierkomplements identisch zu folgendem Ausdruck.
   $$\sim b = 2^l - 1 - b \tag{4.11}$$
   Hierbei wird ein Bitvektor der Länge l, der ausschließlich mit Einsen gefüllt ist $(1...1_{[l]} = 2^l - 1)$, genommen und durch Subtraktion mit b alle zu flippenden Bits auf 0 gesetzt.

2. **Zweier Komplement Negation:**
   Um im Zweierkomplement eine Zahl b zu negieren, werden alle Bits von b geflippt und um 1 addiert.
   $$-b = \sim b + 1 \tag{4.12}$$

3. **Ergebnis:**
   Die Formel (4.11) wird umgeformt, indem beide Seiten mit 1 addiert werden.
   $$\sim b + 1 = (2^l - 1 - b) + 1 \tag{4.13}$$
   $$\sim b + 1 = 2^l - b \tag{4.14}$$

[op]	aussagenlogische Kombination
$[\neq]$	$\neg[=]$
$[\leq]$	$[<] \vee [=]$
$[>]$	$\neg([<] \vee [=])$
$[\geq]$	$\neg[<]$

Tabelle 4.1: *Tabelle zur Kombination relationaler Operatoren*

Das Ergebnis von $\sim b + 1$ ist $2^l - b$. Modulo 2^l angenommen ergibt das $-b$, somit die Negation von b. Durch die Zunahme von mod 2^l kann auf Folgendes geschlossen werden.

$$\langle \sim b \rangle_s + 1 = -\langle b \rangle_s \bmod 2^l \tag{4.15}$$

Insgesamt zeigt dies, dass die Negation eines Bitvektors bv des Zweierkomplements mit $\sim bv + 1$ geformt werden kann.

- **Relationale Operatoren**
  Eine Gleichheit wie $a =_{[l]} b$ wird mithilfe der folgenden Konkatenation implementiert. [2, S. 159]

$$\bigwedge_{i=0}^{l-1} ((a_i \iff b_i) \iff e(t)) \tag{4.16}$$

  Die Ungleichheit wird am Einfachsten mittels Negierung einer Gleichheitsbedingung implementiert.
  Eine Relation $a < b$ wird in den Ausdruck $a - b < 0$ umgeformt. [2, S. 159] Dazu wird ein Addierer für eine Subtraktion, wie vorhin beschrieben, für die umgeformte Relation aufgestellt. [2, S. 159] Weil das Ergebnis von $a < b$ abhängig von der Codierung ist, muss die Handhabung der Codierung jeweils unterschiedlich sein. [2, S. 159] Wenn $cout$ des Addierers nicht gesetzt ist, gilt $a < b$ für vorzeichenlose Operanden in Binärcodierung.

$$\langle a \rangle_u < \langle b \rangle_u \iff \neg add(a, \sim b, 1).cout \tag{4.17}$$

  Im Gegensatz dazu muss bei vorzeichenbehafteten Operanden im Zweierkomplement für $a < b$ $a_{l-1} = b_{l-1} \neq cout$ gelten. Bei der Betrachtung eines einzelnen Bits verhält sich $\oplus$ identisch zu $\neq$.

$$\langle a \rangle_s < \langle b \rangle_s \iff (a_{l-1} \iff b_{l-1}) \oplus add(a, \sim b, 1).cout \tag{4.18}$$

  Vergleiche zwischen Bitvektoren unterschiedlicher Codierung kann mithilfe einer Erweiterung beider Operanden implementiert werden. [2, S. 159] Diese bewirkt, dass beide Bitvektoren danach im Zweierkomplement codiert sind und anschließend ein Vergleich vorzeichenbehafteter Operanden durchführbar ist. [2, S. 159] Die anderen relationalen Operatoren können als Kombination von = und < implementiert werden. [2, S. 149] In Tabelle 4.1 steht [op] für die generierte Bedingung eines Operators op.

- **Verschiebungen**
  Die Links- und Rechtsverschiebung wird beschränkt, sodass die Breite l des verschobenen Bitvektors eine Zweierpotenz und die Breite n des Bitvektors zur Distanz der Verschiebung $\log_2 l$ entspricht. [2, S. 159] Durch diese Beschränkungen kann die Verschiebung beider Richtungen als Barrel-Shifter implementiert werden. [2, S. 159] Die Idee dahinter ist es, die Distanz der Verschiebung bitweise auf den Bitvektor anzuwenden. Dabei entstehen n Stufen, die in einer Stufe s jedes Bit des Vektors um 2^s Bits verschieben kann oder dieses unverändert belässt. [2, S. 159] Nun werden die Funktionen ls (Linksverschiebung) und rs (Rechtsverschiebung) rekursiv für $s \in \{-1,...,n-1\}$ definiert. [2, S. 159] Falls das Bit b_s des zweiten Operanden gesetzt ist, wird das Bit des ersten Operanden verschoben. Wenn das Bit des ersten Operanden an einer Stelle liegt, die aus dem Bitvektor aufgrund der Distanz ohnehin herausgeschoben wird, bricht die rekursive Verschiebung ab und ersetzt die Stelle entsprechend. Ein Barrel-Shifter hat eine sehr geringe Komplexität von $O(n \log n)$ im Vergleich zu $O(n^2)$ bei einer ineffizienten Implementierung der Verschiebung. [2, S. 159]

$$ls(a_{[l]}, b_{[n]u}, -1) \doteq a \tag{4.19}$$

$$ls(a_{[l]}, b_{[n]u}, s) \doteq \lambda i \in \{0,...,l-1\}. \begin{cases} (ls(a,b,s-1))_{i-2^s} & : i \geq 2^s \wedge b_s \\ (ls(a,b,s-1))_i & : i \geq 2^s \wedge \neg b_s \\ 0 & : \text{sonst} \end{cases} \tag{4.20}$$

$$rs(a_{[l]}, b_{[n]u}, -1) \doteq a \tag{4.21}$$

$$rs(a_{[l]}, b_{[n]u}, s) \doteq \lambda i \in \{0,...,l-1\}. \begin{cases} (rs(a,b,s-1))_{i+2^s} & : i < l - 2^s \wedge b_s \\ (rs(a,b,s-1))_i & : i < l - 2^s \wedge \neg b_s \\ (a_{l-1}/0) & : \text{sonst (s / u)} \end{cases} \tag{4.22}$$

- **Multiplikation / Division**
  Die Implementierung der Multiplikation folgt einer sehr simplen Schaltung, die ausschließlich Verschiebung und Addition benötigt. [**keylist**] Zunächst wird die Funktion mul rekursiv für $s \in \{-1,...,n-1\}$ definiert. [2, S. 160] Dabei stellt n die Bitbreite des zweiten Operanden dar. [2, S. 159] Die Idee dahinter ist, die Multiplikation in eine Addition aus Multiplikationen von Zweierpotenzen und a umzuwandeln. Bei einem gesetzten Bit b_s des zweiten Operanden wird a nach links verschoben, sodass a dadurch mit 2^s multipliziert wird.

$$mul(a,b,-1) \doteq 0 \tag{4.23}$$

$$mul(a,b,s) \doteq mul(a,b,s-1) + (b_s ? (a \ll s) : 0) \tag{4.24}$$

Zur Implementierung einer Division $a \div_u b$ werden folgende zwei Bedingungen benötigt.

$$b \neq 0 \implies e(t) * b + r = a \tag{4.25}$$

$$b \neq 0 \implies r < b \tag{4.26}$$

Bei der Implementierung einer vorzeichenbehafteten Division $a \div_s b$ wird folgende zweite Bedingung benötigt.

$$b \neq 0 \implies (r_{l-1} \iff a_{l-1}) \wedge r < |b| \tag{4.27}$$

5 SAT-Solving

Als SAT-Solving wird der Prozess des Lösens eines SAT-Problems mithilfe eines SAT-Solvers bezeichnet.

5.1 SAT

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) ist von großer Bedeutung in der theoretischen Informatik, da dies als erstes NP-vollständiges Problem entdeckt wurde. [9] Bei SAT wird entschieden, ob eine aussagenlogische Formel in konjunktiver Normalform (CNF) erfüllt werden kann. [9] Eine Formel ist genau dann erfüllbar, wenn sie, bei Belegung aller Variablen mit Wahrheitswerten, wahr wird. [9] Um die Zeit solcher Entscheidungen zu verkürzen, wurden SAT-Solver zum Lösen von SAT-Problemen entwickelt. Dabei handelt es sich bei SAT-Solvern um Programme, die effiziente Algorithmen, wie etwa den Davis-Putnam-Logemann-Loveland-Algorithmus (DPLL) oder das Conflict-Driven Clause Learning (CDCL), zur Lösung dieser Probleme nutzen. [9] Mit fortschreitender Entwicklung wurden SAT-Solver noch effizienter, sodass stetig komplexere Probleme schneller gelöst werden können. [11]

5.2 SMT

Aufgestellte Formeln, die nicht interpretierte Funktionen, Differenzlogik oder auch Bitvektorarithmetik benutzen, können nicht von einem SAT-Solver ohne Anwendung von Entscheidungsprozeduren gelöst werden. [11] Hierfür wird allgemein ein SMT-Solver, wie etwa Z3, verwendet. [11] SMT steht für Erfüllbarkeitsmodulo-Theorien, wie beispielsweise zu Bitvektoren fester Weite, Gleichwertigkeit, Arithmetik und Arrays. [11] Ähnlich zu SAT-Problemen wird bei SMT-Problemen die Erfüllbarkeit mithilfe von SMT-Solvern entschieden. [11] Dabei kann der SMT-Solver die Formel des SMT-Problems in ein SAT-Problem zunächst umwandeln und danach lösen. Die Entscheidungsprozedur zu Bitvektoren ist ebenfalls eine SMT-Prozedur und kann demnach die Erfüllbarkeit von Formeln der Bitvektorarithmetik entscheiden. [11]

6 Flattening Erweiterungen

6.1 Inkrementelles Bit Flattening

Die erhaltenen Bedingungen einer Bitvektorformel können für manche Operatoren sehr lang werden. [2, S. 160] Beispielsweise wird in Tabelle 6.1 die Anzahl an Variablen und erstellten CNF Klauseln durch Tseitin-Transformation einer n-bit Multiplikation gezeigt. Zusätzlich dazu wird in Tabelle 6.2 die verbrauchte Zeit bei Tests an 200 n-bit Operationen von verschiedenen Operatoren angegeben. Dabei fällt auf, dass eine Verschiebung schneller als eine Addition gelöst werden kann. Zudem ist die Multiplikation deutlich langsamer als die Addition und würde beim Testen von 64-bit Operationen extrem lange dauern, sodass der Test zu 64-bit Multiplikationen ausgelassen wurde.

n	Anzahl an Variablen	Anzahl an Klauseln
8	313	1001
16	1265	4177
24	2857	9529
32	5089	17057
64	20417	68929

Tabelle 6.1: *Größe der Bedingung einer n-bit Multiplikation nach Tseitin-Transformation [2, S. 160]*

n	Addition	Verschiebung	Multiplikation
4	0,1851s	0,0480s	6,7302s
8	0,3951s	0,0870s	6,7732s
16	0,8367s	0,1758s	31,0054s
32	1,9806s	0,3911s	144,6184s
64	4,4363s	0,8317s	

Tabelle 6.2: *Die verbrauchte Zeit für 200 n-bit Operationen*

Zusätzlich zum Umfang der Formeln einer Multiplikation stellen ihre Symmetrie und Vernetzung eine Belastung für die Entscheidungsheuristik moderner SAT-Solver dar. [2, S. 160]

6.1.1 Vorgehen

Zum Beispiel wird folgende Bitvektorformel genommen:

$$a * b = c \wedge b * a \neq c \wedge x < y \wedge x > y \tag{6.1}$$

Die eben definierte Formel ist offensichtlich nicht erfüllbar, da die ersten zwei Konjunktionen und die letzten zwei Konjunktionen jeweils inkonsistent sind. Es werden

Entscheidungen zu a, b und c begünstigt, da die Entscheidungsheuristiken der meisten SAT-Solver dazu tendieren, zuerst häufig verwendete Variablen aufzuteilen. [2, S. 161] Dementsprechend versuchen SAT-Solver die Unerfüllbarkeit der Formel anhand des komplexen Teils (hier: die Multiplikationen) zu zeigen. [2, S. 161] Dabei wird der leichter zu entscheidende Teil (hier: die Relationen) vernachlässigt. [2, S. 161] Die meisten SAT-Solver können eine derartige Formel langer Bitvektoren nicht in annehmbarer Zeit lösen. [2, S. 161] Allgemein sind arithmetische Operatoren komplexer als bitweise Operatoren. [2, S. 161]

Deshalb ist es häufig sinnvoll, die geflattende Formel mittels inkrementellem Flattenings aufzubauen. [2, S. 161] Das aussagenlogische Skelett wird, wie in Sektion 4.2 beschrieben, zu Beginn aufgestellt. Dem Skelett werden die Bedingungen der unkomplizierten Operatoren hinzugefügt, wobei die Bedingungen der aufwendigen Operatoren zunächst weggelassen werden. [2, S. 161] Es handelt sich beim inkrementellen Bit Flattening um ein Verfahren zur Verfeinerung von Abstraktionen. [2, S. 161] Bei einem Flattening, dem Bedingungen fehlen, handelt es sich um eine Abstraktion der original verwendeten Bitvektorformel. [2, S. 161] Die aktuell geflattende Abstraktion wird dem SAT-Solver zur Lösung gegeben. [2, S. 161] Wenn das Flattening nicht erfüllbar ist, gilt die Unerfüllbarkeit ebenso für die ursprüngliche Bitvektorformel und kann demnach ohne Berücksichtigung der Multiplikationen entschieden werden. [2, S. 161] Sollte hingegen das Flattening erfüllbar sein, tritt einer der folgenden Fälle auf.

1. Die ursprüngliche Bitvektorformel ist nicht erfüllbar und die fehlenden Bedingungen müssen die Erfüllbarkeit der Formel zeigen.
2. Die ursprüngliche Bitvektorformel ist erfüllbar.

Zur Unterscheidung der oben genannten Fälle wird die Belegung der Variablen vom Ergebnis des SAT-Solvers genommen und geprüft, ob diese die weggelassenen Bedingungen erfüllt. Sollte es im fehlenden Teil neue Variablen geben, werden diese einer Konstante zugewiesen und der Belegung von Variablen hinzugefügt. [2, S. 161] Der zweite Fall tritt ein, wenn die Belegung der Variablen alle Bedingungen der Bitvektorformel erfüllt. [2, S. 161] Dies führt zu der Beendigung des Algorithmus. [2, S. 161] Anderenfalls müssen ein oder mehrere Terme, deren Bedingung weggelassen wurde, inkonsistent sein. [2, S. 161] Dabei wählt der Algorithmus einige dieser Terme aus und fügt deren Bedingungen dem Flattening hinzu, um den Algorithmus erneut zu durchlaufen. [2, S. 161] Mit jeder Iteration werden fehlenden Bedingungen dem Flattening hinzugefügt. [2, S. 161] Der Algorithmus endet endgültig, sollte es keine fehlenden Bedingungen mehr geben. [2, S. 161]

6.1.2 Funktionale Konsistenz

Oftmals resultiert das Entfernen der Bedingungen bestimmter Operatoren von Bitvektorformeln in schwachen geflattenden Formeln, wodurch zu viele Belegungen, die im weiteren Verlauf unerfüllbar sind, erfüllt werden. [2, S. 162] Hingegen könnte das Flattening aller Bedingungen den SAT-Solver zu stark beanspruchen. [2, S. 162] Um eine möglichst starke Gesamtbedingung zu erhalten, werden die Bitvektoroperationen mit

nicht interpretierten Funktionen ausgetauscht und diese daraufhin zu Gleichungen umgewandelt. [2, S. 162] Dabei muss die funktionale Konsistenz der Operationen beibehalten werden. [2, S. 162] Es sind zwei Terme $a_1 op b_1$ und $a_2 op b_2$ mit Operator op definiert. [2, S. 162] Zuerst wird op mit einem neuen nicht interpretiertem Funktionssymbol G substituiert. [2, S. 162] Daraufhin wird die Ackermann Reduktion angewandt, um G zu eliminieren. [2, S. 162] Dabei wird jedes Auftreten von $G(a_1, b_1)$ mit einer neuen Variable g_1 und $G(a_2, b_2)$ mit einer neuen Variable g_2 ausgewechselt. [2, S. 163] Schlussendlich wird die Bedingung zur funktionalen Konsistenz der Formel hinzugefügt. [2, S. 163]

$$a_1 = a_2 \wedge b_1 = b_2 \implies g_1 = g_2 \tag{6.2}$$

Die Bedingungen des Flattenings von op sind nicht der resultierenden geflattenden Formel enthalten. [2, S. 163] Die zugefügten Gleichheiten müssen natürlicherweise auch geflattened werden. [2, S. 163]

6.2 Festkommaarithmetik

6.2.1 Aufbau

Die Verwendung der Festkommaarithmetik über Gleitkommaarithmetik stellt einen sinnvollen Kompromiss zwischen Genauigkeit und Komplexität dar. [2, S. 165] Bei der Festkommaarithmetik wird ein Bitvektor einer Zahl in einen ganzzahligen Teil und einen Bruchteil aufgeteilt. [2, S .165] Dabei ist der ganzzahlige Teil vor dem Punkt beziehungsweise Komma und der Bruchteil dahinter. [2, S. 166] Der Begriff Festkommazahl kommt von der festen Anzahl an Ziffern des Bruchteils. [2, S. 165 f.]

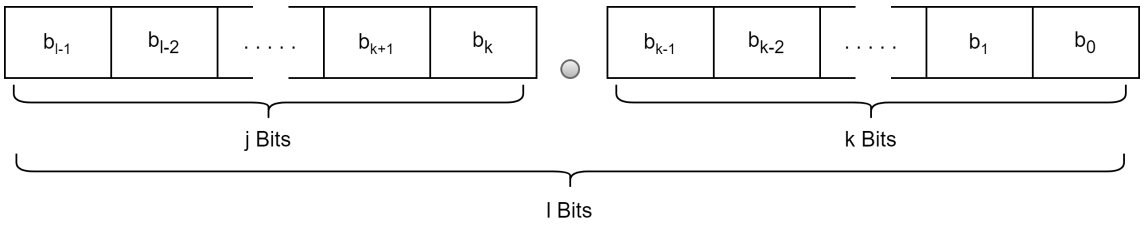

Abbildung 6.1: *Festkomma Bitvektor der Länge $l = j + k$*

Umwandlung der Bitvektoren:
Im Folgenden sind die Bitvektoren M, F mit ihrer entsprechende Bitlänge m, f gegeben. [2, S. 166]

$$\langle \cdot \rangle : \{0,1\}^{m+f} \to Q \tag{6.3}$$

$$\langle M.F \rangle \doteq \frac{\langle M \circ F \rangle_s}{2^f} \tag{6.4}$$

Codierungsbeispiele:

$$\langle 011.01 \rangle = 3,25 \tag{6.5}$$

$$\langle 110.1 \rangle = -1,5 \tag{6.6}$$

Einige rationale Zahlen, wie die periodische Zahl $\frac{1}{3}$ können in der Festkommaarithmetik, aufgrund von Limitierungen der Bitvektoren, nur angenähert werden. [2, S. 166] Im Gegensatz zum Überlauf gibt es auch einige Varianten der Festkommaarithmetik, die eine Sättigung implementieren. [2, S. 166] Anstatt eines Umbruchs behält das Ergebnis dabei den höchsten oder niedrigsten möglichen Wert bei. [2, S. 166] Die hier definierte Variante der Festkommaarithmetik verwendet die Überlaufsemantik. Beispielsweise kann folgendermaßen eine Addition von zwei Bitvektoren (a und b), die als Festkommazahl codiert sind, definiert werden.

$$a_M.a_F + b_M.b_F = c_M.c_F \iff \langle a_M.a_F \rangle \cdot 2^f + \langle b_M.b_F \rangle \cdot 2^f = \langle c_M.c_F \rangle \cdot 2^f \bmod 2^{m+f} \tag{6.7}$$

6.2.2 Flattening

Das Flattening auf die Festkommaarithmetik kann ebenso gut angewendet werden wie auf die bei Binärcodierung oder dem Zweierkomplement. [2, S. 167] Dabei wird davon ausgegangen, dass die Seiten links und rechts des binären Operators die selbe Anzahl an Bits, vor und nach dem Komma, besitzen. [2, S. 167] Sollte dies nicht der Fall sein, können fehlende Bits hinter dem Komma hinzugefügt werden, indem der Bruchteil mit Nullen von rechts gepolstert (Zero-Padding) wird. Hingegen werden fehlende Bits vor dem Komma von links mithilfe einer Vorzeichenerweiterung angefügt. [2, S. 167] Das Flattening der bitweisen Operatoren funktioniert exakt gleich und kann demnach für die Festkommaarithmetik übernommen werden. [2, S. 167] Ebenso können die Bedingungen zu Addition, Subtraktion und relationale Operatoren, wie bei den Binärzahlen vorhin, definiert werden. [2, S. 167] Bei der Multiplikation hingegen besitzt der Bruchteil des Ergebnisses zweier Zahlen, mit x und y Bits jeweils im Bruchteil, x + y Bits im Bruchteil. Meist werden weniger Bits als verfügbar benötigt. [2, S. 167] Diese überschüssigen Bits können mithilfe einer zusätzlichen Rundung im Ergebnis eliminiert werden. [2, S. 167]

konkrete Beispiele zur Addition zweier Festkommazahlen

$$\langle 001.11 \rangle + \langle 000.01 \rangle = \langle 010.00 \rangle \tag{6.8}$$

$$\langle 000.0 \rangle + \langle 1.10 \rangle = \langle 111.10 \rangle \tag{6.9}$$

$$\langle 01.0 \rangle + \langle 110.0 \rangle = \langle 111.0 \rangle \tag{6.10}$$

konkrete Beispiele zur Multiplikation zweier Festkommazahlen

$$\langle 00.1 \rangle * \langle 01.1 \rangle = \langle 0.11 \rangle \tag{6.11}$$

$$\langle 01.10 \rangle * \langle 01.1 \rangle = \langle 10.010 \rangle \tag{6.12}$$

$$\langle 0.11 \rangle * \langle 101.0 \rangle = \langle 110.000 \rangle \tag{6.13}$$

Zur Verringerung der Länge des Bruchteils kann eine Rundung (zu Null, zur nächsten geraden Zahl oder zu [+ / -] ∞) ebenfalls angewandt werden. [2, S. 167]

7 Implementierung

Die Implementierung der Entscheidungsprozedur zu Bitvektoren ist Teil des existierenden Projekts **haskell-smt** von Matthias Güdemann. Dabei handelt es sich um ein GitLab-Projekt, das in Haskell diverse SMTs realisiert. Derzeit gibt es Entscheidungsprozeduren zu nicht interpretierten Funktionen und der Differenzlogik. Mithilfe des Projekts können Terme der Aussagenlogik erstellt und diese in Formen, wie Negationsnormalform (NNF) und konjunktive Normalform (CNF), umgewandelt werden. Das Projekt ermöglicht auch die Tseitin-Transformationen und kann mithilfe eines SAT-Solvers die Erfüllbarkeit aussagenlogischer Terme (in CNF) prüfen. Insgesamt enthält das Projekt alle essentiellen Komponenten zur Implementierung einer Entscheidungsprozedur für Bitvektoren in Haskell.

7.1 Vorgehen

Im ersten Schritt wurde eine umfassende Analyse des Projekts durchgeführt, um ein grundlegendes Verständnis über die bestehende Codebasis aufzubauen. Dies ist insofern wichtig, da viele der benötigten Elemente bereits implementiert waren und demnach nicht erneut entwickelt werden mussten. Darauf aufbauend wurde ein minimaler Funktionsumfang der finalen Implementierung programmiert, um die wichtigsten Anforderungen des Projekts festzulegen. Hierzu gehörte ebenfalls eine vorläufige Definition der neu benötigten Datenstrukturen. Nach dem ersten groben Entwurf der Struktur wurde diese dann finalisiert, um anschließend die Funktionalität für alle Operatoren einer Bitvektorformel schrittweise zu programmieren. Dabei wurden zunächst einfache und später komplexe Operatoren gewählt. Abschließend sind Tests zur Validierung der Programmierung und zur Entscheidungsprozedur mit Bitvektoren erstellt worden.

7.2 Datentypen

- **Bitvektor**
  In Haskell gibt es mehrere Pakete (engl. packages) zur Erstellung und Verwendung von Bitvektoren. Die Implementierung verwendet eine Kombination aus **vector** und **bitvec**. Ebenfalls kann dafür **bv**, **bv-little** oder **array** verwendet werden, da diese ähnliche Pakete, wie etwa **bitvec**, sind. [3] Die einfachste Möglichkeit ist es einen Vektor / ein Array aus Bool Werten als Bitvektor zu verwenden. Da es in Paketen wie **bitvec** oder **bv** Funktionen zu Bitvektoroperationen und Bit basierten Operationen gibt, werden diese bevorzugt. [3] Dabei gibt es Unterschiede zwischen **bv** und **bitvec**.

```
type BitVec = U.Vector Bit
```

Abbildung 7.1: *Datentyp für Bitvektoren*

So ist **bitvec** im Vergleich zu **bv** besser gewartet, da dort die letzten Änderungen neuer sind. Bei **bv** sind die letzten Änderungen älter. [1] Zudem ist die Verwendung von **bitvec** deutlich effizienter für den Speicher (8 mal geringerer Speicherbedarf) und ist für bestimmte Bitoperationen 3500 mal schneller im Vergleich zu Vektoren aus Bool Werten. [3] Mithilfe des Pakets **vector** können Vektoren, die eine effiziente Implementierung von Int-indizierten Arrays sind, erstellt und darauf zugegriffen werden. [8]

- **Bitvektorformel** Die Implementierung der Datentypen zur Darstellung von Bitvektorformeln basiert auf der in Sektion 3.3 definierten Grammatik. Die Haskell Typen wurden mithilfe des Statements *data* erstellt. Dabei sind mehrere Konstruktoren einem Datentyp zugewiesen, die mit dem Zeichen | unterteilt werden. [6] Das erste Argument ist der jeweilige Name eines Datentyps und die folgenden Argumente sind eine Liste an Variablen eines Datentyps. [6] Zudem können Datentypen ebenfalls ohne zusätzlich benötigte Variablen definiert werden. [6] Besonders nützlich ist der Typ beim Mustervergleich (engl. pattern matching), der in Haskell oft bei Funktionen Anwendung findet.

```
data BVFormula = BoolConj BVFormula BVFormula
               | BoolNeg BVFormula
               | BVAtom BVAtom
               deriving (Eq, Ord, Show)
```

Abbildung 7.2: *Datentyp für formula*

In BVAtom steht BVTop für True und BVBottom für False als Wahrheitswert. Bei der Implementierung des Atoms wurde die Ableitung zu *term rel term* nicht gleichartig in einem Haskell Datentyp definiert. Dementsprechend musste dieser Teil in einen eigenen Typ, der Relationen zwischen Bitvektoren darstellt, auslagert werden.

```
data BVAtom = BVRel BVRel
            | BVTop
            | BVBottom
            | BVAt BVTerm BitVec
            deriving (Eq, Ord, Show)
```

Abbildung 7.3: *Datentyp für atom*

Zur eigentlichen Implementierung der Relation zweier Bitvektoren wird für jeden relationalen Operator ein eigener Konstruktor definiert.
In Abbildung 7.5 steht BVNeg für den ~ Operator und stellt die bitweise Negation eines BVTerms dar. BVTernary steht für den ternären Operator. Der Konstruktor hat ein BVAtom und zwei BVTerm als Variablen, was insgesamt zu der Ableitung eines Terms zu *atom?term:term* führt. Der Konstruktor von BVRange steht, wie in der Grammatik beschrieben, für *term[constant:constant]*. Die Konstanten sind in diesem

```
data BVRel = BVLess BVTerm BVTerm
           | BVGreater BVTerm BVTerm
           | BVLessEqual BVTerm BVTerm
           | BVGreaterEqual BVTerm BVTerm
           | BVEqual BVTerm BVTerm
           | BVNotEqual BVTerm BVTerm
           deriving (Eq, Ord, Show)
```

Abbildung 7.4: *Datentyp für term rel term*

Fall Bitvektoren des Typs BitVec, die als vorzeichenlose, binärcodierte Bitvektoren fungieren. Um Bitvektoren als Identifikator oder Konstante zu definieren, mussten die Konstruktoren BVIdentifier und BVConstant, die beide jeweils eine Codierung benötigen, angewendet werden. Beim Identifikator wurde zudem Name und Länge des Bitvektors angegeben. Dabei muss, ähnlich zur Implementierung des Atoms, auch bei BVTerm die Ableitung zu *term op term* der Grammatik in einen neuen Haskell Datentyp für Bitvektoroperationen ausgelagert werden.

```
data BVTerm = BVOp BVOp
            | BVIdentifier BVEncoding String Int
            | BVNeg BVTerm
            | BVConstant BVEncoding BitVec
            | BVTernary BVAtom BVTerm BVTerm
            | BVRange BVTerm BitVec BitVec
            | BVExtension BVTerm BitVec
            deriving (Eq, Ord, Show)
```

Abbildung 7.5: *Datentyp für term*

Der Datentyp BVOp für Bitvektoroperationen implementiert zu jedem Operator der Bitvektorarithmetik jeweils einen Konstruktor.

```
data BVOp = Concat BVTerm BVTerm
          | Xor BVTerm BVTerm
          | Or BVTerm BVTerm
          | And BVTerm BVTerm
          | RightShift BVTerm BitVec
          | LeftShift BVTerm BitVec
          | Division BVTerm BVTerm
          | Multiplication BVTerm BVTerm
          | Subtraction BVTerm BVTerm
          | Addition BVTerm BVTerm
          deriving (Eq, Ord, Show)
```

Abbildung 7.6: *Datentyp für term op term*

Die Codierung wurde ebenfalls als Datentyp BVEncoding implementiert, da diese für Identifikatoren und Konstanten von Bitvektoren benötigt wird.

```haskell
data BVEncoding = BinaryEncoding
                | Two'sComplement
                deriving (Eq, Ord, Show)
```

Abbildung 7.7: *Datentyp für Codierung*

- **Status für Hilfsvariablen**
  Um neue einzigartige Hilfsvariablen, die beim Flattening einer Bitvektorformel benötigt werden, zu erstellen, muss es sich erhöhende Variablen geben, auf die während des Flattenings zugegriffen werden kann. Dafür wird für die Implementierung eine global veränderbare Variable benötigt. In Haskell ist das mithilfe von **globalVar = unsafePerformIO (newMVar var)** möglich. [5] Die Variable verwendet MVar als ihren Datentyp, da dieser eingebaute Synchronisation bereitstellt und somit eine sichere Nutzung bietet. [4] Außerdem lassen sich leicht zusätzliche Funktionen, die den Status verwenden, implementieren. So lässt sich der Erhalt der Hilfsvariablennamen sicherstellen und die Modifizierung des Status erhöhen.

```haskell
-- State Monad für die Benennung der Hilfsvariablen
type VariableState = Int
type VariableMonad = StateT VariableState IO

startVarState :: VariableState
startVarState = 0 :: VariableState

-- globaler VariableState
{-# NOINLINE globalVarState #-}
globalVarState :: MVar VariableState
globalVarState = unsafePerformIO (newMVar startVarState)
```

Abbildung 7.8: *Datentyp für den Variablenstatus*

7.3 Funktionen

Zu Beginn der funktionalen Implementierung wurde eine Validierung für Bitvektorformeln (BVFormula) erstellt. Die Funktion muss ebenfalls für alle anderen Haskell Datentypen einer Bitvektorformel implementiert werden. Dabei wird geprüft, ob die Länge und Codierung der Bitvektoren für die jeweils angewandten Operationen valide sind. Beispielsweise muss die Länge beider Bitvektoren einer Addition oder die Codierung bei relationalem Operator gleich sein. Um möglichen auftretenden Problemen beim Flattening zu entgehen, sollte die Bitvektorformel zunächst validiert werden.

```haskell
-- validiert die Formel
validateFormula :: BVFormula -> Bool
```

Abbildung 7.9: *Validierung von BVFormula*

Aufgrund der globalen Eigenschaft des Variablenstatus wird er zu Beginn des Flattenings mit **resetVarState** auf seinen Startwert zurückgesetzt. Wird dies nicht gemacht,

erhöht sich der Variablenstatus nach jeder Anwendung des Flattenings. Das Zurücksetzen wird nicht zwingend benötigt. Trotzdem verschafft es einen besseren Überblick bei der Anzeige einer geflattenden Bitvektorformel, da die booleschen Variablennamen aufgrund des kleineren Status kürzer sind. Die Funktion zum Flattening muss ebenfalls für alle Datentypen einer Bitvektorformel implementiert werden. Das Ergebnis der Funktion musste zum Typ IO Term gesetzt werden, da innerhalb der Funktion auf die globale Variable zugegriffen werden muss. Dabei kann der Zugriff nur innerhalb eines Haskell do-Blocks durchgeführt werden. Zum Ende erfolgt die Rückgabe des Terms mittels **return**, das als IO Funktion implementiert ist.

```haskell
-- Start von Flattening einer Bitvektorformel
bvFlattening :: BVFormula -> IO Term
bvFlattening f = do
  stateStart <- evalGlobalVariableMonad resetVarState
  bvFlatteningFormula f stateStart
```

Abbildung 7.10: *Flattening mit Reset des Variablenstatus*

```haskell
-- Flattening einer Bitvektorformel
bvFlatteningFormula :: BVFormula -> VariableState -> IO Term
```

Abbildung 7.11: *Flattening von BVFormula*

Abbildung 7.12 zeigt wie die Implementierung vom Flattening eines Operators grundsätzlich aufgebaut ist. Der Variablenname des Ergebnisses wird durch die Anwendung von **varNameFromState** auf den übergebenen Status erhalten. Um neue Hilfsvariablen und deren korrespondierenden Status zu erstellen, wird **generateNewVarState** angewendet. Da eine Operation & aus zwei Termen besteht, muss auf beide Terme die Flattening Funktion angewendet werden. Insgesamt erfolgt die Rückgabe einer Konkatenation der geflattenden Terme und der generierten Bedingung, die hier in **term** zwischengespeichert wird. Die dabei erstellten Bedingungen basieren auf der Flattening Theorie aus Sektion 4.3. Mithilfe der Funktion **getBVarAt** aus Abbildung 7.13 wird die boolesche Variable des Bits eines Terms erhalten. Dabei handelt es sich bei dem Term entweder um einen Identifikator, eine Konstante oder einen anderen Term. Im Falle eines Identifikators wird der Name des Identifikators mit dem Index des gewünschten Bits kombiniert. Bei einer Konstanten wird mittels Vektorzugriff das gewünschte Bit als True oder False Wahrheitswert zurückgegeben. Ansonsten wird der Hilfsvariablenname mit dem Bit Index wie bei dem Identifikator zusammengestellt.

```haskell
bvFlatteningOp (And a b) state = do
  varRes <- evalVariableMonad varNameFromState state
  (varOne, stateOne) <- runGlobalVariableMonad generateNewVarState
  (varTwo, stateTwo) <- runGlobalVariableMonad generateNewVarState
  fstTerm <- bvFlatteningTerm a stateOne
  sndTerm <- bvFlatteningTerm b stateTwo
  let term = foldl1 (/\) [getBVarAt a varOne n /\ getBVarAt b varTwo n <-> BVar (varRes ++ "_" ++ show n) | n <- [0..(lengthTerm a - 1)]]
  return (term /\ fstTerm /\ sndTerm)
```

Abbildung 7.12: *Flattening des bitweisen Operators &*

```haskell
-- gibt entweder den Wert des BVTerms oder die BVar an einer Stelle zurück
getBVarAt :: BVTerm -> String -> Int -> Term
getBVarAt (BVIdentifier _ str _) _ n = BVar (str ++ "_" ++ show n)
getBVarAt (BVConstant enc bitvec) _ n =
  case bitvec U.! n of
    Bit True -> Top
    Bit False -> Bottom
getBVarAt _ varName n = BVar (varName ++ "_" ++ show n)
```

Abbildung 7.13: *Bit an einer Stelle eines Terms erhalten*

a	b	$a \vee b$	$\neg(a \wedge b)$	$(a \vee b) \wedge \neg(a \wedge b)$	$a \oplus b$
T	T	T	F	F	F
T	F	T	T	T	T
F	T	T	T	T	T
F	F	F	T	F	F

Tabelle 7.1: *Wahrheitstabelle zu XOR*

Ein Volladdierer benötigt zwei Terme als Bedingung. Zunächst den **termCarry**, der die Bedingung aller Übertragbits des Volladdierers festlegt und den **termRes**, der die Bedingung für alle Bits des Ergebnisses beinhaltet. Um auf das Übertragsbit der Ausgabe zugreifen zu können, wird das Bit am Index der Länge von einem der beiden Terme referenziert. Dabei kann durch Anwendung von **varNameCarryFromState** der Name des Übertragbits erhalten werden. Alle normalen Hilfsvariablen werden mit **e + varState + _ index** benannt. Bei Übertragbits wird statt einem **e** ein **c** zur Differenzierung benutzt. Da der $\oplus$ (XOR) Operator nicht in der Aussagenlogik des Projekts implementiert ist, muss $a \oplus b$ zu $(a \vee b) \wedge \neg(a \wedge b)$ umgewandelt werden. Diese beiden aussagenlogische Terme sind gleichwertig, da sich die Wahrheitstabellenwerte beider Terme aus Tabelle 7.1 gleichen. Aufgrund der vielen $\oplus$ Operationen im Volladdierer und deren Umwandlungen werden die Bedingungen insgesamt länger, was sich negativ auf die Leserlichkeit auswirkt. Das Flattening des < Operators ist etwas effizienter gestaltet, da dort anstatt der Bedingung eines Volladdierers ausschließlich die der Übertragbits verwendet wird. Bei dem relationalen Operator < wird ausschließlich das **cout** Bit benötigt, weshalb die Bedingung für das Ergebnis des Volladdierers weglassen werden kann.

In Abbildung 7.15 wird die Theorie zum Flattening einer Verschiebung aus Sektion 4.4 implementiert. Insbesondere zeigt sich die Rekursion beim Flattening durch die unterschiedlichen Fälle, wie beispielsweise **s == -1**.

```
-- Flattening eines Full Adders für Addition und Subtraktion
flatteningAddition :: BVOp -> Bit -> VariableState -> IO Term
flatteningAddition (Addition a b) cin state = do
  let l = lengthTerm a
  varRes <- evalVariableMonad varNameFromState state
  varCarry <- evalVariableMonad varNameCarryFromState state
  (varOne, stateOne) <- runGlobalVariableMonad generateNewVarState
  (varTwo, stateTwo) <- runGlobalVariableMonad generateNewVarState
  fstTerm <- bvFlatteningTerm a stateOne
  sndTerm <- bvFlatteningTerm b stateTwo
  let termCin = case cin of
        Bit True -> BVar (varCarry ++ "_" ++ show 0) <-> Top
        Bit False -> BVar (varCarry ++ "_" ++ show 0) <-> Bottom
  -- carry(a,b,c) <=> (a ∧ b) ∨ ((a ⊕ b) ∧ c)
  let termCarry = termCin /\ foldl1 (/\) [(((getBVarAt a varOne n /\ getBVarAt b varTwo n) \/ (((getBVarAt a varOne n \/ getBVarAt b varTwo n) /\
  -- sum(a,b,c) <=> (a ⊕ b) ⊕ c
  let termRes = foldl1 (/\) [(((getBVarAt a varOne n \/ getBVarAt b varTwo n) /\ negation (getBVarAt a varOne n /\ getBVarAt b varTwo n)) \/ BVa
  return (termCarry /\ termRes /\ fstTerm /\ sndTerm)
```

Abbildung 7.14: *Flattening des Volladdierers*

```
-- Flattening des Barrel Shifters zur Verschiebung nach links
flatteningLeftShift :: BVTerm -> BitVec -> Int -> Int -> String -> Term
flatteningLeftShift a b s i varOne = term
  where
    term
      | s == -1 = getBVarAt a varOne i
      | b U.! s == Bit True = if i >= 2^s
                              then flatteningLeftShift a b (s - 1) (i - 2^s) varOne
                              else Bottom
      | otherwise = flatteningLeftShift a b (s - 1) i varOne
```

Abbildung 7.15: *Flattening des Barrel-Shifters*

7.4 Anwendung

Im Folgenden wird das ausführende Beispiel 3.18 aus Sektion 3.3 schrittweise auf die Implementierung in Haskell angewendet.

1. **Konstruktion:**
   Die Konstruktion einer Bitvektorformel verwendet die in Sektion 7.2 eben definierten Datentypen. Hier steht **formula** für die Bitvektorformel des ausführenden Beispiels als Datentyp **BVFormula** in Haskell.
2. **Validierung:**
   Vor dem Flattening sollte **formula** noch validiert werden. Im Beispiel ist die **formula** valide, da die Formel **validateFormula** als Ergebnis True anzeigt.
3. **Flattening:**
   Um das Flattening einer Bitvektorformel wie **formula** durchzuführen, muss die

```
-- Flattening der Multiplikation
flatteningMultiplication :: BVTerm -> BVTerm -> Int -> BVTerm
flatteningMultiplication a b s = term
  where
    zeroBitVector = U.replicate (lengthTerm a) (Bit False)
    sBitVec = intToBitVec 0 s
    term
      | s == -1 = BVConstant BinaryEncoding zeroBitVector
      | otherwise = BVOp (Addition (flatteningMultiplication a b (s - 1)) (BVTernary (BVAt b sBitVec) (BVOp (LeftShift a sBitVec)) (BVConstant BinaryEncoding zeroBitVector)))
```

Abbildung 7.16: *Flattening der Multiplikation*

```
ghci> formula
BVAtom (BVRel (BVEqual (BVOp (Addition (BVOp (Xor (BVConstant BinaryEncoding [0,0,1,1]) (BVConstant BinaryEncoding [0,1,0,1])))
(BVConstant BinaryEncoding [0,1,1,0]))) (BVIdentifier BinaryEncoding "result" 4)))
```

Abbildung 7.17: *Konstruktion der Bitvektorformel*

```
ghci> validateFormula formula
True
```

Abbildung 7.18: *Validierung der Bitvektorformel*

Funktion **bvFlattening**, die in Abbildung 7.10 definiert wurde, Anwendung finden. Zudem kann am resultierenden Ausdruck der Aussagenlogik erkannt werden, wie eine geflattende Bitvektorformel strukturiert ist.

```
ghci> flattenedTerm <- bvFlattening formula
ghci> flattenedTerm
('"e0"' /\ (((((('"e1_0"' <-> '"result_0"') /\ ('"e1_1"' <-> '"result_1"')) /\ ('"e1_2"' <-> '"result_2"')) /\ ('"e1_3"' <-> '"result_3"')) <->
'"e0"') /\ (((((('"c1_0"' <-> F ) /\ ((((((('"e3_0"' /\ F ) \/ (('"e3_0"' \/ F ) /\ ~ ('"e3_0"' /\ F )) /\ '"c1_0"')) <-> '"c1_1"') /\ ((('"e3
_1"' /\ T ) \/ ((('"e3_1"' \/ T ) /\ ~ ('"e3_1"' /\ T )) /\ '"c1_1"')) <-> '"c1_2"')) /\ ((('"e3_2"' /\ T ) \/ ((('"e3_2"' \/ T ) /\ ~ ('"e3
_2"' /\ T )) /\ '"c1_2"')) <-> '"c1_3"')) /\ ((('"e3_3"' /\ F ) \/ ((('"e3_3"' \/ F ) /\ ~ ('"e3_3"' /\ F )) /\ '"c1_3"')) <-> '"c1_4"'))) /\
 (((((((('"e3_0"' \/ F ) /\ ~ ('"e3_0"' /\ F )) \/ '"c1_0"') /\ ~ ((('"e3_0"' \/ F ) /\ ~ ('"e3_0"' /\ F )) /\ '"c1_0"')) <-> '"e1_0"') /\ ((
(((('"e3_1"' \/ T ) /\ ~ ('"e3_1"' /\ T )) \/ '"c1_1"') /\ ~ ((('"e3_1"' \/ T ) /\ ~ ('"e3_1"' /\ T )) /\ '"c1_1"')) <-> '"e1_1"')) /\ (((((('"
e3_2"' \/ T ) /\ ~ ('"e3_2"' /\ T )) \/ '"c1_2"') /\ ~ ((('"e3_2"' \/ T ) /\ ~ ('"e3_2"' /\ T )) /\ '"c1_2"')) <-> '"e1_2"')) /\ (((((('"e3_3"
' \/ F ) /\ ~ ('"e3_3"' /\ F )) \/ '"c1_3"') /\ ~ ((('"e3_3"' \/ F ) /\ ~ ('"e3_3"' /\ F )) /\ '"c1_3"')) <-> '"e1_3"'))) /\ (((((((( F \/
F ) /\ ~ ( F /\ F )) <-> '"e3_0"') /\ ((( F \/ T ) /\ ~ ( F /\ T )) <-> '"e3_1"')) /\ ((( T \/ F ) /\ ~ ( T /\ F )) <-> '"e3_2"')) /\ (
(( T \/ T ) /\ ~ ( T /\ T )) <-> '"e3_3"')) /\ T ) /\ T )) /\ T ))
```

Abbildung 7.19: *Flattening der Bitvektorformel*

4. **SAT-Solving:**
   Bei Anwendung von **extractResult** wird die **flattennedFormula** zuerst in CNF-Form gebracht. Dabei werden Tseitin-Transformationen und eine Vereinfachung des Ausdruckes ausgeführt. Mithilfe von **solveCNFIO**, einer Methode aus dem SAT Modul des Projekts, wird das aufgestellte SAT-Problem gelöst. Intern verwendet die Methode dabei den SAT-Solver **PicoSAT**. Schlussendlich wird der Identifikator **result** des Ergebnisbitvektors aus der Liste aller Variablen der SAT-Solver Lösung gefiltert und ein boolescher Wert für jedes Bit des Ergebnisbitvektors zurückgegeben. Sollte dabei nicht der gewünschte Bitvektor Identifikator aus der Lösung gefiltert werden, so ist das Ergebnis eine sehr lange Liste an Variablen und deren angenommenen booleschen Werte, wie in Abbildung 7.22 zu sehen. Die so entstandene Liste enthält überwiegend Hilfsvariablen aus Tseitin-Transformationen und Flattenings, die dem Endergebnis keine nützlichen Informationen hinzufügen und es dadurch unleserlicher machen.

```
ghci> extractResult flattenedFormula
Just [("result_0",False),("result_1",False),("result_2",True),("result_3",True)]
```

Abbildung 7.20: *Ergebnis der geflattenden Bitvektorformel*

```haskell
extractResult :: Term -> BVResult
extractResult f =
  case simplifyCNF $ tseitin f of
    Nothing -> error "not in CNF"
    Just c  -> do
      solution <- solveCNFIO c
      let result = fmap (getResultVars . cleanAuxVars . Map.toList) solution
      pure result
```

Abbildung 7.21: *Funktion zum SAT-Solving einer geflattenden Bitvektorformel*

```
ghci> solveCNFIO c
Just (fromList [("aux10",True),("aux101",False),("aux104",False),("aux105",True),("aux106",False),("aux108",False),("aux109",True),("aux11",True),("aux110",False),("aux112",True),("aux115",True),("aux118",True),("aux119",False),("aux120",False),("aux122",False),("aux125",True),("aux128",True),("aux129",False),("aux130",False),("aux132",False),("aux133",True),("aux134",False),("aux136",True),("aux137",True),("aux14",True),("aux140",True),("aux143",True),("aux144",False),("aux145",False),("aux147",True),("aux15",True),("aux150",True),("aux153",True),("aux154",False),("aux155",False),("aux157",False),("aux158",True),("aux159",True),("aux161",True),("aux162",True),("aux165",False),("aux168",False),("aux169",True),("aux17",True),("aux170",False),("aux172",True),("aux175",False),("aux178",False),("aux179",True),("aux180",False),("aux182",False),("aux183",True),("aux184",True),("aux186",True),("aux187",True),("aux188",True),("aux191",False),("aux194",False),("aux195",True),("aux196",False),("aux198",True),("aux20",True),("aux201",True),("aux204",False),("aux205",True),("aux206",True),("aux208",True),("aux209",True),("aux212",True),("aux215",False),("aux216",True),("aux217",True),("aux219",True),("aux220",True),("aux223",True),("aux226",True),("aux227",False),("aux228",False),("aux23",False),("aux230",True),("aux231",True),("aux233",True),("aux235",True),("aux236",True),("aux238",True),("aux239",True),("aux241",True),("aux242",True),("aux26",False),("aux29",False),("aux3",True),("aux30",True),("aux31",False),("aux33",False),("aux34",False),("aux36",True),("aux39",True),("aux42",True),("aux45",True),("aux46",False),("aux47",False),("aux49",False),("aux50",True),("aux52",True),("aux53",True),("aux56",True),("aux59",True),("aux6",True),("aux62",True),("aux63",False),("aux64",False),("aux66",False),("aux67",True),("aux69",True),("aux7",True),("aux70",True),("aux73",False),("aux76",False),("aux79",False),("aux80",True),("aux81",False),("aux83",False),("aux84",False),("aux86",True),("aux87",True),("aux88",True),("aux91",False),("aux94",False),("aux95",True),("aux96",False),("aux98",False),("c1_0",False),("c1_1",False),("c1_2",True),("c1_3",True),("c1_4",False),("e0",True),("e1_0",False),("e1_1",False),("e1_2",True),("e1_3",True),("e3_0",False),("e3_1",True),("e3_2",True),("e3_3",False),("result_0",False),("result_1",False),("result_2",True),("result_3",True)])
```

Abbildung 7.22: *Ergebnis beim SAT-Solving ohne Filterung*

Prüfung:
Gemäß dem allgemeinen Vorgehen wird abschließend das Ergebnis auf Korrektheit geprüft. Dafür wird $(1100 \oplus_4 1010) +_4 0110$ mit dem Ergebnis aus Abbildung 7.20 verglichen.

$$1100 \oplus_4 1010 = 0110 \tag{7.1}$$

$$0110 +_4 0110 = 1100 \tag{7.2}$$

Da das Prüfergebnis dem Ergebnis aus Abbildung 7.20 gleicht, ist somit das Ergebnis korrekt.

8 Fazit

In dieser Arbeit wurde eine Entscheidungsprozedur für Bitvektorarithmetik entwickelt, um der Fehleranfälligkeit von Computersystemen entgegenzuwirken. Durch die Erarbeitung der Theorie konnte eine praktische Anwendung dafür in Haskell umgesetzt werden.

Insgesamt wurden alle eingangs gesetzten Ziele vollständig erfüllt. Die in Haskell vollständig abgebildeten Bitvektorformeln konnten in Verbindung mit Bitflattening in aussagenlogische Formeln umgewandelt werden. Aufbauend auf der anschließenden Umwandlung mittels Tseitin-Transformationen in konjunktive Normalform konnte die Formel mit SAT-Solving gelöst werden. Die abschließende Prüfung auf Korrektheit des Ergebnisses aus Sektion 7.4 bestätigt die Erreichung der Ziele.

Allgemein sind SMT-Solver, wie die Entscheidungsprozedur für Bitvektoren, Teil der formalen Verifikation von Schaltkreisen, Programmen und Methoden der Hardware- und Softwareentwicklung. [11] Die formale Verifikation ist dabei essentiell, um die Korrektheit der Systeme zu beweisen. Das Ergebnis der Arbeit leistet insofern einen Beitrag die Korrektheit von Systemen zu beweisen, indem es zur formalen Verifikation verwendet werden kann.

Dennoch gibt es einige Einschränkungen in der vorliegenden Arbeit. So können beispielsweise ausschließlich Bitvektoren mit Binärcodierung oder Zweierkomplement in den Bitvektorformeln verwendet werden. Andere Bitvektorcodierungen können somit nicht zur Erstellung von Formeln der Bitvektorarithmetik angewendet werden. Zudem entstehen bei Operationen mit großen Bitvektoren, wie der Multiplikation, extrem komplexe aussagenlogische Formeln, die nur schwer von einem SAT-Solver in annehmbarer Zeit gelöst werden können.

Als Ausblick könnte das Projekt **haskell-smt** möglicherweise um weitere SMTs erweitert werden. Dafür würden beispielsweise Arrays oder Induktive Datentypen in Frage kommen. [11] Zudem wäre die Kombination unterschiedlicher schon bestehender Theorien eine mögliche Erweiterung. [11] Ebenso könnte die Entscheidungsprozedur zur Bitvektorarithmetik, wie in Kapitel 6 beschrieben, erweitert werden, sodass diese andere Codierungen von Bitvektoren unterstützt oder optional ein inkrementelles Flattening anwendet.

Abbildungsverzeichnis

Tabellenverzeichnis

Hilfsmittel

Chat-GPT: Dies wurde unterstützend, vor allem in Teilen der Arbeit wie Einleitung und Fazit, zum Verfassen verwendet.

Literatur

[1] I. Abal. *bv: Bit-vector arithmetic library*. 7. Okt. 2024. URL: https://hackage.haskell.org/package/bv.

[2] „Bit Vectors“. Englisch. In: *Decision Procedures. An Algorithmic Point of View*. 2008.

[3] *bitvec: Space-efficient bit vectors*. 7. Okt. 2024. URL: https://hackage.haskell.org/package/bitvec.

[4] T. U. of Glasgow 2001. *Control.Concurrent.MVar*. 7. Okt. 2024. URL: https://hackage.haskell.org/package/base-4.20.0.1/docs/Control-Concurrent-MVar.html.

[5] HaskellWiki. *Top level mutable state*. 7. Okt. 2024. URL: https://wiki.haskell.org/Top_level_mutable_state.

[6] HaskellWiki. *Type*. 7. Okt. 2024. URL: https://wiki.haskell.org/Type.

[7] R. C. Lacher. *Bits and Bytes*. 7. Okt. 2024. URL: https://www.cs.fsu.edu/~lacher/courses/COP3330/lectures/bitvectors/script.html.

[8] R. Leshchinskiy. *vector: Efficient Arrays*. 7. Okt. 2024. URL: https://hackage.haskell.org/package/vector.

[9] Mbssdr. *Erfüllbarkeitsproblem der Aussagenlogik*. 7. Okt. 2024. URL: https://de.wikipedia.org/wiki/Erf%C3%BCllbarkeitsproblem_der_Aussagenlogik.

[10] Nomen4Omen. *Bitkette*. 21. Aug. 2024. URL: https://de.wikipedia.org/wiki/Bitkette.

[11] *Satisfiability Modulo Theories*. 7. Okt. 2024. URL: https://theory.stanford.edu/~barrett/pubs/BSST21.pdf.