Institut für theoretische Informatik

Leibniz Universität Hannover

Bachelorarbeit

Ein Programm für affine modallogische Formeln

Timon Barlag
Matrikelnummer: 3077970

4. August 2016

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier
Betreuer: M. Sc. Maurice Chandoo

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst habe und keine anderen Hilfsmittel und Quellen als angegeben verwendet habe.

_____

Timon Barlag

Inhaltsverzeichnis

1 Einleitung

Die theoretische Informatik beschäftigt sich mit einer Vielzahl von unterschiedlichen Inhalten. Neben Themen wie der Komplexitätstheorie oder den formalen Sprachen ist auch die Logik ein viel behandelter Teil der theoretischen Informatik. Eines der wichtigsten Probleme der Logik ist das sogenannte Erfüllbarkeitsproblem der Aussagenlogik SAT, welches fragt, ob eine übergebene aussagenlogische Formel erfüllbar ist oder nicht. Damit verwandt ist das etwas weniger bekannte Erfüllbarkeitsproblem der Modallogik ML-SAT.

Die Modallogik selbst ist ein relativ altes Konzept, für welches Ansätze schon in Begriffen wie der von Leibniz geprägten „möglichen Welt " zu finden sind. Im Wesentlichen erweitert sie die Aussagenlogik um die modalen Begriffe „möglich" ($\Diamond$) und „notwendig" ($\Box$) und bietet so zusätzliche Möglichkeiten, logische Aussagen zu charakterisieren. So erlaubt sie beispielsweise für Zustände eines Programms mögliche und notwendige Eigenschaften von zukünftigen Zuständen zu beschreiben. Definiert ist die Modallogik über sogenannte Kripke-Rahmen, welche wiederum in unterschiedliche Rahmenklassen unterteilt sind. Diese Rahmenklassen separieren Kripke-Rahmen anhand ihrer Eigenschaften. Die Rahmenklasse T beispielsweise enthält alle reflexiven Kripke-Rahmen, also alle Kripke-Rahmen, in denen jede Welt in ihren eigenen Nachfolgerwelten enthalten ist. Während SAT in NP liegt, ist ML-SAT PSPACE-vollständig [Ladner 1977]. Aufgrund dieser erhöhten Komplexität wurde in [Hemaspaandra et al. 2010] die Komplexität von sogenannten booleschen Fragmenten der Modallogik untersucht. Diese Fragmente bestehen jeweils aus einer Menge von booleschen Funktionen und gegebenenfalls Konstanten in Kombination mit den Teilmengen der modallogischen Operatoren $\{\Box, \Diamond\}$. Besondere Schwierigkeiten macht hier unter anderem das Fragment $\{\oplus, \top\}$, also das „exklusive Oder" mit der Konstanten „wahr" in den Rahmenklassen T, $\mathsf{S4}$ und $\mathsf{S5}$.

Diese Arbeit beschäftigt sich mit der Frage nach der Existenz bestimmter Formeln über dem zuvor genannten Fragment der Modallogik. Explizit setzt sich diese Arbeit mit der Suche nach zwei erfüllbaren modallogischen Formeln φ und ψ unterschiedlicher modaler Tiefe auseinander, sodass $\varphi \oplus \psi$ unerfüllbar ist. Diese Formeln werden in Kripke-Rahmen der Rahmenklassen T und $\mathsf{S4}$ gesucht.

Aufgrund des wenig intuitiven Verhaltens von $\oplus$ steht in dieser Arbeit die Erstellung eines Programms im Vordergrund, welches systematisch nach Formeln solcher Art sucht.

Diese Arbeit ist wie folgt strukturiert. Zu Beginn wird in Kapitel 2 die Modallogik in der Form, wie sie hier verwendet wird, eingeführt. Danach wird in Kapitel 3 zunächst die Programmstruktur erläutert und anschließend werden die einzelnen Programmteile vorgestellt. Darauf folgend werden in Kapitel 4 die Ergebnisse, die mit dem Programm erzielt wurden, vorgestellt und zum Schluss folgt in Kapitel 5

eine Zusammenfassung und ein Ausblick basierend auf den Ergebnissen dieser Arbeit.

2 Grundlagen der Modallogik

„Modal languages are simple yet expressive languages for talking about relational structures. [...]
Modal languages provide an internal, local perspective on relational structures. [...]
Modal languages are not isolated modal systems.“

- [Blackburn et al. 2001]

Die Modallogik erweitert die bekannten Wahrheitsbegriffe der Aussagenlogik „wahr“ und „falsch“ um die modalen Begriffe „möglich“ und „notwendig“. Zusätzlich zu den aussagenlogischen Operatoren enthält die Modallogik damit die modalen Operatoren $\Diamond$ und $\Box$, mit

$\Diamond\varphi \approx$ „φ ist möglicherweise wahr“ und
$\Box\varphi \approx$ „φ ist notwendigerweise wahr“

für aussagenlogische Formeln φ.
Hierfür gelten dann auch $\Diamond\varphi \equiv \neg\Box\neg\varphi$ und $\Box\varphi \equiv \neg\Diamond\neg\varphi$.

Die nachfolgenden Ausführungen basieren zum Großteil auf dem Skript Logik und formale Systeme von Vollmer und Kluge aus dem Sommersemester 2015. [Vollmer & Kluge 2015]

2.1 Semantik

Die Semantik der Modallogik ist definiert über sogenannte Kripke-Rahmen. Ein Kripke-Rahmen ist ein Paar $F = (W, R)$ mit

W ist eine nichtleere Menge von Welten und

R ist eine binäre Relation auf W.

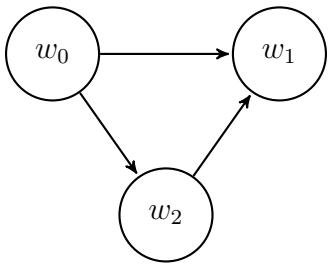

Abbildung 2.1: *Graphische Repräsentation von Kripke-Rahmen* $F = (\{w_0, w_1, w_2\}, \{(w_0, w_1), (w_0, w_2), (w_2, w_1)\})$

Eine Kripke-Struktur oder ein Modell der Modallogik ist ein Paar $K = (F, V)$ mit:

$F = (W, R)$ ist ein Kripke-Rahmen

$V : Var \to \mathcal{P}(W)$ ist eine Abbildung, die jeder Variablen p eine Menge $V(p)$ von Welten zuordnet. In diesen Welten ist p wahr.
(Var bezeichnet hier die Menge aller Variablen, $\mathcal{P}(W)$ bezeichnet die Potenzmenge von W)

Für eine Kripke-Struktur $K = (F, V)$ mit Kripke-Rahmen $F = (W, R)$ wird die Kurzschreibweise $K = (W, R, V)$ benutzt.

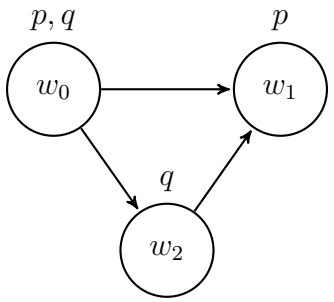

Abbildung 2.2: *Graphische Repräsentation von Kripke-Struktur* $K = (\{w_0, w_1, w_2\}, \{(w_0, w_1), (w_0, w_2), (w_2, w_1)\}, \{p \to w_0, p \to w_1, q \to w_0, q \to w_2\})$

2.2 Erfüllbarkeit

Für aussagenlogische Formeln φ und ψ, ein Modell $M = (W, R, V)$ und eine Welt $w \in W$ schreiben wir $M, w \models \varphi$, falls φ in Welt w erfüllt ist.

Es gilt:

$M, w \models \top$	immer,
$M, w \models \bot$	nie,
$M, w \models p$	falls $w \in V(p)$,
$M, w \models \neg\varphi$	falls nicht $M, w \models \varphi$,
$M, w \models \varphi \wedge \psi$	falls $M, w \models \varphi$ und $M, w \models \psi$,
$M, w \models \varphi \vee \psi$	falls $M, w \models \varphi$ oder $M, w \models \psi$,
$M, w \models \Diamond\varphi$	falls für mindestens ein $v \in W$ mit $(w, v) \in R$ gilt, dass $M, v \models \varphi$,
$M, w \models \Box\varphi$	falls für alle $v \in W$ mit $(w, v) \in R$ gilt, dass $M, v \models \varphi$

Für diese Arbeit ist zusätzlich noch erwähnenswert, dass gilt:

$$M, w \models \varphi \oplus \psi \quad \text{falls} \quad \begin{array}{ll} \text{entweder} & M, w \models \varphi \text{ und nicht } M, w \models \psi \\ \text{oder} & \text{nicht } M, w \models \varphi \text{ und } M, w \models \psi \end{array}$$

Wir nennen eine modale Formel φ erfüllbar, wenn es eine Kripke-Struktur $M = (W, R, V)$ und eine Welt $w \in W$ gibt, sodass $M, w \models \varphi$.
Die Formel $\varphi = \Diamond\Diamond p \oplus \Box q$ ist beispielsweise erfüllbar, da sie in der Kripke-Struktur K aus Abbildung 2.2 in der Welt w_0 erfüllt ist, also $K, w_0 \models \varphi$ gilt.

2.3 Rahmenklassen

Kripke-Rahmen in der Modallogik können verschiedene Eigenschaften haben. Um sie dementsprechend zu klassifizieren, werden sie in Rahmenklassen unterteilt.
Für diese Arbeit wichtig sind vor allem die Rahmenklassen T und $\mathsf{S4}$.

T – reflexive Rahmen

T ist die Klasse aller reflexiven Kripke-Rahmen, das heißt aller Kripke-Rahmen $F = (W, R)$ mit der Eigenschaft, dass R reflexiv ist.
Ein Rahmen F ist also genau dann in T, wenn für alle Formeln φ gilt:

$$F \models \varphi \to \Diamond\varphi$$

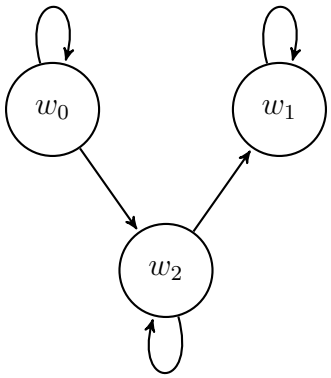

Abbildung 2.3: *Graphische Repräsentation von reflexivem Kripke-Rahmen* $F = (\{w_0, w_1, w_2\}, \{(w_0, w_0), (w_0, w_2), (w_1, w_1), (w_2, w_1), (w_2, w_2)\})$

S4 – **reflexive und transitive Rahmen**

S4 ist die Klasse aller reflexiven und transitiven Kripke-Rahmen, das heißt aller Kripke-Rahmen $F = (W, R)$ mit der Eigenschaft, dass R transitiv und reflexiv ist. Ein Rahmen F ist also genau dann in S4, wenn für alle Formeln φ gilt:

$$F \models \Diamond\Diamond\varphi \to \Diamond\varphi \quad \text{und} \quad F \models \varphi \to \Diamond\varphi$$

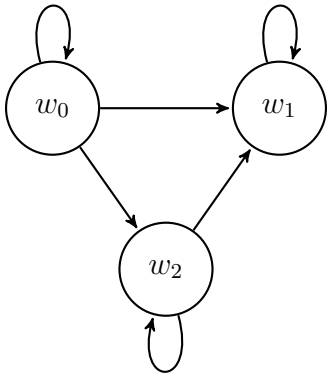

Abbildung 2.4: *Graphische Repräsentation von reflexivem und transitivem Kripke-Rahmen* $F = (\{w_0, w_1, w_2\}, \{(w_0, w_0), (w_0, w_1), (w_0, w_2), (w_1, w_1), (w_2, w_1), (w_2, w_2)\})$

2.4 Modale Tiefe

Die modale Tiefe einer modallogischen Formel bezeichnet die größte Anzahl an modalen Operatoren, also $\Diamond$ und $\Box$, die auf eine einzelne atomare Formel angewendet werden. Sie repräsentiert also die maximale Schachtelungstiefe der modalen Operatoren einer Formel.

Die modale Tiefe einer Formel φ wird bezeichnet mit $md(\varphi)$. Sei p eine atomare Formel, also $p \in Var \cup \{\top, \bot\}$ mit Var der Menge aller Variablen. Seien außerdem φ und ψ modallogische Formeln. Dann ist die modale Tiefe wie folgt definiert:

$$\begin{aligned} md(p) &:= 0, \\ md(\neg p) &:= 0, \\ md(\varphi \vee \psi) &:= max(md(\varphi), md(\psi)), \\ md(\Diamond\varphi) &:= md(\varphi) + 1 \end{aligned}$$

Für die Belange dieser Arbeit sind außerdem folgende Fälle erwähnenswert:

$$\begin{aligned} md(\varphi \oplus \psi) &:= max(md(\varphi), md(\psi), \\ md(\Box\varphi) &:= md(\varphi) + 1 \end{aligned}$$

Des Weiteren wird in dieser Arbeit eine Änderung an der obigen Definition vorgenommen. Im Folgenden kennzeichnet die modale Tiefe nur die größte Anzahl an modalen Operatoren, die auf eine Variable angewendet werden. Damit gilt also für $c \in \{\top, \bot\}$:

$$\begin{aligned} md(\Diamond c) &:= 0, \\ md(\Box c) &:= 0, \\ md(\varphi \oplus \psi) &:= \begin{cases} 0, & \text{falls } \varphi \in \{\top, \bot\} \text{ und } \psi \in \{\top, \bot\}, \\ max\{md(\varphi), md(\psi)\}, & \text{sonst} \end{cases} \end{aligned}$$

Folglich gilt also beispielsweise:

$$\begin{aligned} md(x \oplus \Diamond y) &= 1 \\ md(\Box(x \oplus \Diamond y)) &= 2 \\ md(\Box(x \oplus \Diamond \top)) &= 1 \end{aligned}$$

3 Programm

3.1 Aufbau und Struktur

Das im Verlaufe dieser Arbeit entwickelte Programm ist in unterschiedliche Teile unterteilt, welche zusammenarbeiten, um Formeln der gewünschten Form zu suchen. Eine vereinfachte Visualisierung der Funktionsweise ist in Abbildung 3.1 dargestellt.

Um die modallogischen Formeln, die benutzt werden, möglichst einfach vom Computer bearbeiten zu lassen, werden sie hier in der Form von sogenannten Syntaxbäumen dargestellt. Um diese Repräsentation zu erreichen wird der Programmteil um die Klasse `SyntaxTree` verwendet. Er dient dazu, die Arbeit mit Formeln der Modallogik intern einfacher zu gestalten und Formeln abzuspeichern, also aus einem Syntaxbaum wieder eine Formel in Stringform zu erstellen.

Die Umkehrung davon, also aus einem String einen Syntaxbaum zu erzeugen, ist etwas schwieriger als das Abspeichern eines Syntaxbaumes als String. Daher gibt es dafür die Klasse `RecursiveDescentParser`, welche mithilfe eines Parsers eine valide modallogische Formel in String-Form zu einem Syntaxbaum parst und bei invaliden Eingaben abbricht.

Der Programmteil `TreeGenerator` generiert alle Syntaxbäume mit bestimmten Eigenschaften. Im Programm dient er dazu, alle Formeln bestimmter Eigenschaften zu erzeugen, um diese später zu überprüfen.

Um Kripke-Rahmen und Kripke-Strukturen im Programm darzustellen, wurde die Klasse `KripkeFrame` implementiert. Zusätzlich zur internen Repräsentation eines Kripke-Rahmens dient dieser Teil außerdem zum Prüfen, ob einzelne Formeln in einzelnen Welten des Rahmens erfüllt sind.

Der Programmteil um die Klasse `GraphGenerator` dient dazu, alle nichtisomorphen Graphen mit bestimmten Eigenschaften zu generieren. Diese werden dazu verwendet, alle möglichen Kripke-Rahmen mit diesen Eigenschaften zu erstellen, um später möglichst schnell zu prüfen, ob einzelne Formeln erfüllbar sind oder nicht.

Die Klasse `FrameLabeler` diente ursprünglich dazu, Variablen in Kripke-Rahmen mithilfe eines Backtracking-Algorithmus' so auf die einzelnen Welten zu verteilen, dass eine übergebene Formel in dem Rahmen erfüllt ist. Da der verwendete Algorithmus jedoch zu tief rekursiv gearbeitet und so schon recht schnell „Stackoverflow"-Fehler verursacht hat, wurde hier auf eine simplere Methode zurückgegriffen. Anstelle eines Labeling-Algorithmus' durch Backtracking wurde hier mit der Klasse `BruteForceLabeler` ein Bruteforce-Algorithmus zur Variablenbelegung der Kripkerahmen entwickelt. Dieser dient im Programm dazu, herauszufinden, ob zu einer gegebenen Formel und einem gegebenen Graphen irgendeine Variablenbelegung des Graphen existiert, sodass dieser als `KripkeFrame` die Formel erfüllt.

Der Programmteil um `FormulaGenerator` umfasst den ersten Teil des Zusammenarbeitens der bisher genannten Programmteile mit dem Ziel, alle möglichen modallogischen Formeln bestimmter Größe zu generieren. Da dies bei größeren Formeln ziemlich viel Zeit in Anspruch nehmen kann, wurde durch regelmäßige Abspeicherung aller benötigten Daten auch die Möglichkeit, dass Programm zu unterbrechen und fortzusetzen, implementiert.

Der zweite Teil der Zusammenarbeit aller anderen Programmteile ist der Programmteil um `FormulaChecker`. Dieser dient dazu, mit allen generierten Formeln nach einer Formel der in dieser Arbeit gewünschten Form zu suchen. Ähnlich wie bei `FormulaGenerator` ist auch hier die Möglichkeit implementiert, abzubrechen und wieder fortzusetzen.

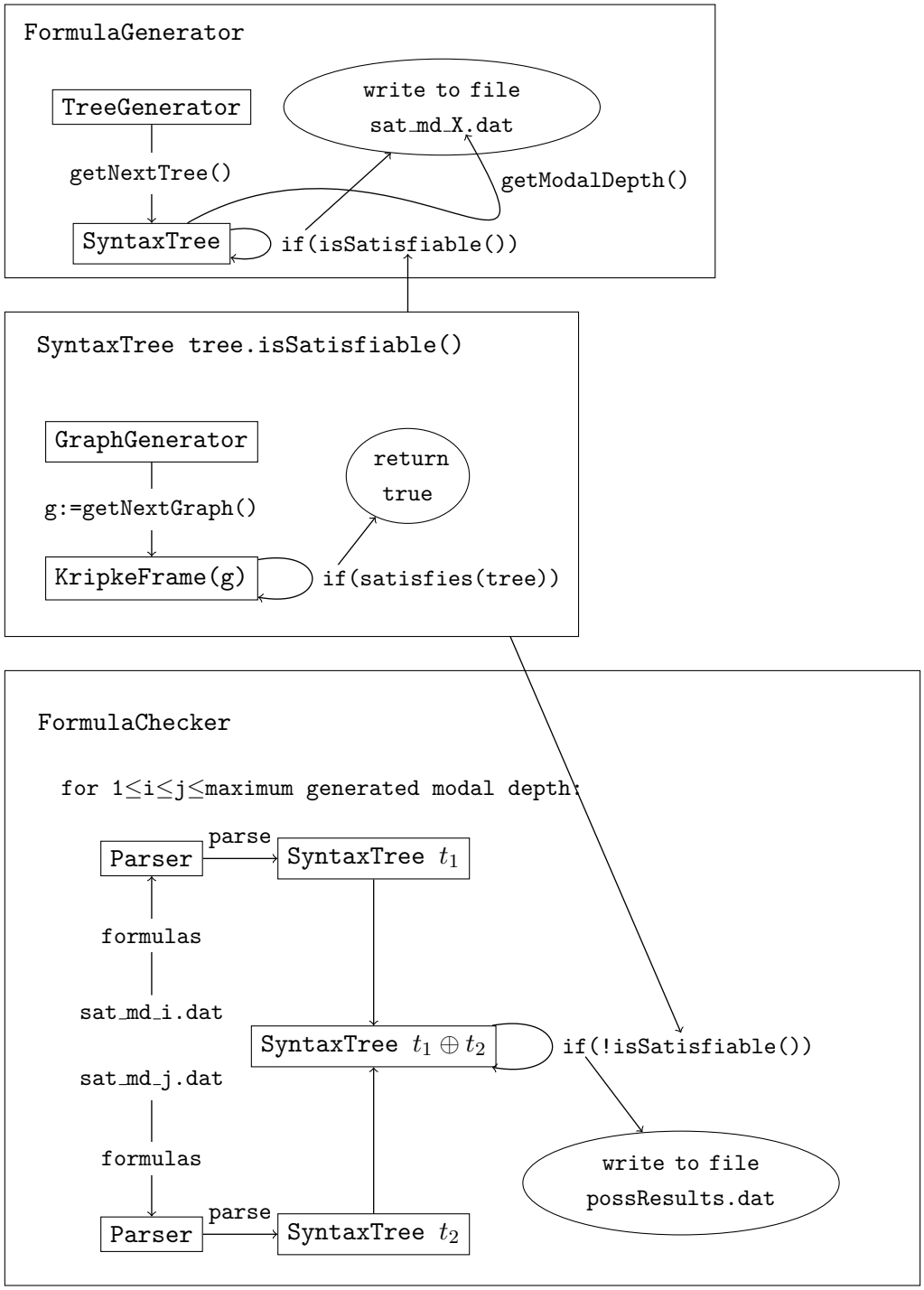

Abbildung 3.1: *Vereinfachte Visualisierung der Funktionsweise des in dieser Arbeit erstellten Programms*

3.2 SyntaxTree

Syntaxbäume

Syntaxbäume, wie sie in dieser Arbeit verwendet werden, sind Bäume, die logische Formeln repräsentieren. Es werden logische Operationen, Konstanten und Variablen als Knoten dargestellt, wobei eine logische Operation jeweils die Knoten als Nachfolger hat, auf welche sie angewendet wird. Konstanten und Variablen sind die Blätter des Baumes.

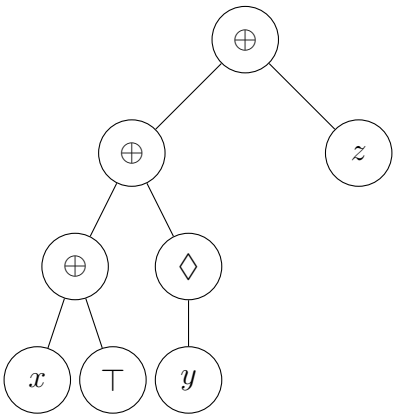

Abbildung 3.2: *Syntaxbaum-Repräsentation der Formel* $x \oplus \top \oplus \Diamond y \oplus z$

Mit Syntaxbäumen lässt sich besser arbeiten als mit Formeln im String-Format, da sie direkt die logische Struktur der Formel repräsentieren und so erlaubten, durch Durchlaufen des Baumes beispielsweise direkt Aussagen über die modale Tiefe einer modallogischen Formel zu machen. Diese ist nämlich hier die größte Anzahl an modalen Operatoren in einem Pfad von der Wurzel des Syntaxbaumes bis zu einem Variablenknoten. In dem obigen Beispiel in Abbildung 3.2 kann man also die modale Tiefe von 1 direkt ablesen.

Implementierung

Syntaxbäume wurden in `SyntaxTree` in Form von Knoten und Nachfolgern implementiert. Ein `SyntaxTree`-Objekt besteht also im Wesentlichen aus einer Referenz auf den Wurzelknoten des Baumes, welcher selbst Referenzen auf seine Nachfolger enthält. Dies macht es vor allem einfach, Teilformeln aus einzelnen Syntaxbäumen zu generieren. Dazu muss lediglich der Knoten des „als letztes auszuführenden Operators“, oder bei einer einelementigen Formel der Knoten der Variablen/Konstanten, als Wurzel gewählt werden. Mit dem „als letztes auszuführenden Operator“ ist hier der Operator gemeint, der am schwächsten bindet

und am weitesten rechts in der Formel steht. Um zum Beispiel in dem Syntaxbaum aus Abbildung 3.2 die Teilformel $x \oplus \top \oplus \Diamond y$ zu erhalten, wählt man als Wurzel den Knoten des $\oplus$ vor dem $\Diamond y$, da zwar alle $\oplus$ gleich stark binden, das $\oplus$ weiter rechts aber durch die Formelauswertung von links zuletzt aufgerufen werden würde.

Um die Syntaxbäume intern darstellen zu können, wurden bestimmte Symbole gewählt, um die einzelnen modallogischen Symbole zu repräsentieren. Die Symbole, die bei dem betrachteten Fragment der Modallogik relevant sind, wurden folgendermaßen zugeordnet:

$\oplus$	wird dargestellt als	$+$
$\Diamond$	wird dargestellt als	$\#$
$\Box$	wird dargestellt als	@
$\top$	wird dargestellt als	1

In der Klasse `SyntaxTree` selber sind als nützliche Methoden `getModalDepth()`, `toFormula()` und `isSatisfiable(int)` implementiert.

Die Methode `getModalDepth()` sucht mit Durchlauf des Baumes durch Tiefensuche den längsten Pfad vom Wurzelknoten bis zu einem Variablenknoten und gibt so die modale Tiefe des Syntaxbaumes zurück.

Die Methode `toFormula()` generiert rekursiv eine zu dem Syntaxbaum äquivalente Formel im String-Format.

Die Methode `isSatisfiable(int)` sucht nach einer Kripke-Struktur, welche den Syntaxbaum erfüllt. Der übergebene Parameter ist die maximale Größe von Kripke-Rahmen, in denen gesucht werden soll. Falls unter den gegebenen Bedingungen keiner gefunden wird, wird `falsch` zurückgegeben, ansonsten wird `wahr` zurückgegeben. Gesucht wird mithilfe der Klasse `GraphGenerator`, welche alle möglichen nicht-isomorphen Graphen mit bestimmten Parametern generiert. Hier wird jeweils die maximale Größe der generierten Graphen eingeschränkt, der größte erlaubte Durchmesser wird auf die modale Tiefe der Formel gesetzt und der größte erlaubte Knotengrad wird auf die Anzahl an Diamanten[1] in der Formel + 1 gesetzt.

Der maximale Durchmesser wird auf die modale Tiefe gesetzt, da die am weitesten von der zu überprüfenden Welt entfernte für die Formel relevante Welt maximal die modale Tiefe von der zu überprüfenden Welt entfernt sein kann. Dies liegt daran, dass nur maximal so viele modale Operatoren auf einzelne Variablen der Formel angewendet werden. Da in dieser Arbeit nur reflexive Kripke-Rahmen betrachtet werden, macht die hier verwendete Abwandlung der modalen Tiefe auch keinen Unterschied, da jede Welt mindestens sich selbst als Nachfolgerwelt hat und somit in allen Welten $\Diamond\top \equiv \top$ gilt.

[1] Für die Suche nach Formeln der der Aufgabenstellung entsprechenden Form wurden hier von den modallogischen Operatoren nur $\Diamond$ in den Bäumen benutzt, da mit $\Box\varphi \equiv \neg\Diamond\neg\varphi$ und $\neg\varphi \equiv \varphi \oplus \top$ alle modalen Formeln, die $\Box$ enthalten, über $\{\oplus, \top\}$ auch ohne $\Box$ dargestellt werden können.

Der maximale erlaubte Knotengrad wird auf die Anzahl an Diamanten in der Formel + 1 gesetzt, da es, um eine Formel $\Diamond\varphi$ zu erfüllen, reicht, wenn es eine Nachfolgerwelt gibt, in der φ erfüllt ist. Damit muss die Anzahl der Nachfolgerwelten einer Welt, abgesehen von sich selbst, maximal gleich der Anzahl der Diamanten sein, die höchstens auf eine Teilformel in ihr angewendet werden können. Dafür ist die Anzahl an Diamanten in der Formel + 1 eine obere Grenze.

Für diese Graphen wird dann jeweils ein Kripke-Rahmen erstellt, für welchen eine Variablenbelegung gesucht wird, die die Formel erfüllt.

3.3 Parser

Für das Parsen von Formeln im String-Format zu Syntaxbäumen wurde ein Top-Down-Parser nach der Methode des rekursiven Abstieges verwendet. Die Funktionsweise und die Erstellung dieses Parsers sind [Parchmann 2011] entnommen.
Für den Parser wurde die folgende LL(1)-Grammatik gewählt:

$$
\begin{aligned}
&\{\{S, S', V\}, \{(,), \Diamond, \Box, var, 1, \oplus\},\\
&\{S \to (S)S',\\
&\ S \to \Diamond S,\\
&\ S \to \Box S,\\
&\ S \to varVS',\\
&\ S \to 1S',\\
&\ S' \to \oplus S,\\
&\ S' \to \varepsilon,\\
&\ V \to varV,\\
&\ V \to \varepsilon\},\\
&S\}
\end{aligned}
$$

Hierbei steht das Terminal 1 für die Konstante $\top$, das Terminal *var* für ein Symbol, das in einem Variablennamen vorkommen kann und das Terminal ε für das leere Wort. Diese Grammatik erzeugt alle Formeln des gewünschten Fragments der Modallogik mit den Einschränkungen, dass ein Variablenname nicht mit 1 beginnen darf und keines der anderen Terminalsymbole enthalten darf. Außerdem muss hier noch eingeschränkt werden, dass das Symbol $ nicht in den Formeln enthalten sein darf, da $ einerseits im Parsing-Algorithmus benötigt wird und andererseits an späterer Stelle verwendet wird, um Formeln sicher in eine Datei abzuspeichern. Im Programm wird der Parser mit der Klasse `RecursiveDescentParser` initialisiert und erlaubt den Aufruf der Methode `parse(String)`, welche einen String übergeben bekommt und, sofern der übergebene String eine für das gewünschte Fragment valide Formel unter den gegebenen Einschränkungen ist, einen repräsen-

tativen Syntaxbaum zurückgibt. Ist der übergebene String keine valide Formel, so wird `null` zurückgegeben.

3.4 TreeGenerator

Die Klasse `TreeGenerator` dient dazu, alle nicht-isomorphen Bäume mit bestimmten Eigenschaften zu erzeugen. Hierfür wird das Programm `gentreeg` aus [nauty & Traces] aus dem Paper [McKay, Piperno 2014] verwendet, welches alle ungerichteten, ungewurzelten, nicht-isomorphen Bäume mit bestimmten Eigenschaften generiert. Der Algorithmus aus Abbildung 3.3 ist eine vereinfachte Form des Algorithmus', der im Programm zur Generierung von für die Belange dieser Arbeit brauchbaren Bäumen verwendet wird.

```
1  Rufe gentreeg mit gewünschten Parametern auf;
2  forall von gentreeg erzeugte, ungerichtete, ungewurzelte Bäume t do
3  |  forall mögliche gewurzelte Varianten t_r von t do
4  |  |  Weise jedem Knoten mit Ausgangsgrad 1 das Symbol ◊ zu und
   |  |   jedem Knote mit Ausgangsgrad 2 das Symbol ⊕;
5  |  |  (Teile die Blattknoten von t_r in Variablen und Konstanten ein)
   |  |   forall Einteilungen der Blattknoten in die Mengen V (Variablen)
   |  |   und K (Konstanten) do
6  |  |  |  forall unterschiedliche Mengenpartitionen P_V = {p_1, ..., p_n} der
   |  |  |   Knoten in V do
7  |  |  |  |  for i = 1,...,n do
8  |  |  |  |  |  weise allen Knoten aus p_i die Variable x_i zu;
9  |  |  |  |  |  return t_r;
10 |  |  |  |  end
11 |  |  |  end
12 |  |  end
13 |  end
14 end
15 return null;
```

Abbildung 3.3: **Algorithmus**: *Generiere brauchbare Bäume (vereinfacht).*

Der Algorithmus in der Implementierung funktioniert analog zum Algorithmus aus Abbildung 3.3 mit dem Unterschied, dass er sich seinen Zustand bei einem `return` merkt und dann beim nächsten Aufruf da weitermacht, wo er zuvor aufgehört hat. Man kann sich vorstellen, dass der Algorithmus im Programm genauso funktioniert wie der Algorithmus aus Abbildung 3.3 mit dem Zusatz, dass er

anstatt bei `return` t_r in Zeile 9 zu terminieren, dort lediglich anhält und beim nächsten Aufruf dort weitermacht.

Um in Zeile 3 alle möglichen gewurzelten Varianten eines ungerichteten, ungewurzelten Baumes zu erhalten, wird über alle Knoten des Baumes iteriert. Da sowohl $\oplus$ als auch $\Diamond$ als Wurzel in Frage kommen, können alle Knoten mit Grad 1 oder 2 als Wurzel angenommen werden. Also wird für jeden solchen Knoten der ungerichtete Baum gewurzelt.

Für die Einteilungen der Blattknoten in Variablen und Konstanten in Zeile 5 wird jede mögliche solche Einteilung vorgenommen. Dies geschieht über das Hochzählen einer Binärzahl, deren Länge gleich der Anzahl der Blattknoten des Baumes ist. Jedem Blattknoten wird dann ein Index zwischen 0 und der Länge der Binärzahl - 1 zugeordnet. Anschließend werden für jeden Wert der Binärzahl von 000...0 bis 111...1 jeweils die Werte an den einzelnen Stellen der Binärzahl ausgelesen. Ist an Stelle i der Wert 0, so wird der Blattknoten mit dem Index i den Variablen zugeordnet. Ist er 1, so wird der Blattknoten den Konstanten zugeordnet.

Um in Zeile 6 alle unterschiedlichen Mengenpartitionen der Variablen zu erhalten, wird der dafür vorgesehene Algorithmus aus [Knuth 2004] verwendet.

3.5 KripkeFrame

Die Klasse `KripkeFrame` dient dazu, Kripke-Rahmen und Kripke-Strukturen im Programm zu repräsentieren. Dies wird im Wesentlichen durch

- einen Graph in Form einer Adjazenzmatrix,
- einer Liste von Variablen und
- einer Map mit den Variablen als Schlüssel und den Welten, in denen sie gesetzt sind, als Werte

realisiert.

Abgesehen von Helfer-Methoden ist in der Klasse `KripkeFrame` noch die Methode `evaluate(SyntaxTree, int)` implementiert. Sie überprüft, ob der übergebene Syntaxbaum in der übergebenen Welt erfüllt ist. Dies geschieht über den Algorithmus aus Abbildung 3.4.

Algorithmus: `evaluate`
Umgebung: Kripke-Struktur $K = (W, R, V)$
Parameter: Syntaxbaumknoten $Node$, Integer $World$

```
1  Definiere Tree := der Syntaxbaum mit Node als Wurzel;
2  Definiere Root := die Wurzel von Tree;
3  switch Symbol von Root do
4      case ◇ do
5          forall Welten w mit (World, w) ∈ R do
6              if evaluate(Nachfolger von Root, w) gibt wahr zurück then
7                  return wahr;
8              end
9          end
10         return falsch;
11     end
12     case □ do
13         forall Welten w mit (World, w) ∈ R do
14             if evaluate(Nachfolger von Root, w) gibt falsch zurück then
15                 return wahr;
16             end
17         end
18         return wahr;
19     end
20     case ⊕ do
21         Definiere N_l := der linke Nachfolger von Root;
22         Definiere N_r := der rechte Nachfolger von Root;
23         if (evaluate(N_l, world) gibt wahr zurück und
24             evaluate(N_r, world) gibt falsch zurück) oder
25             (evaluate(N_l, world) gibt falsch zurück und
26             evaluate(N_r, world) gibt wahr zurück) then
27             return wahr;
28         else
29             return falsch;
30         end
31     end
```

```
32      case Variable do
33          if Root ist gesetzt in world then
34              return wahr;
35          else
36              return falsch;
37          end
38      end
39      case Konstante do
40          if Root ist wahr then
41              return wahr;
42          else
43              return falsch;
44          end
45      end
46  end
```

Abbildung 3.4: **Algorithmus**: Prüfe, ob Formel φ, repräsentiert durch Syntaxbaum $Tree$ in Kripke-Struktur K erfüllt ist.

3.6 GraphGenerator

Implementierung

Die Klasse `GraphGenerator` dient dazu, alle nicht-isomorphen Graphen mit bestimmten Eigenschaften zu generieren. Diese werden dazu verwendet, Kripke-Rahmen zu erzeugen, um später herauszufinden, ob bestimmte modallogische Formeln erfüllbar sind. Um diese Graphen zu generieren, werden die Programme `geng` und `watercluster2` aus [nauty & Traces] aus dem Paper [McKay, Piperno 2014] verwendet. `geng` generiert nicht-isomorphe Graphen mit bestimmten Eigenschaften und `watercluster2` generiert alle nicht-isomorphen gerichteten Versionen dieser Graphen mit bestimmten Eigenschaften. Insgesamt funktioniert die Graphgenerierung von `GraphGenerator` nach folgendem Algorithmus:

```
1  Rufe geng mit den gewünschten Parametern auf;
2  while es gibt noch unbenutzte, von geng generierte Graphen g_g do
3  |  Rufe watercluster2 mit den gewünschten Parametern auf und
   |   übergebe g_g;
4  |  while es gibt noch unbenutzte, von watercluster2 generierte Graphen
   |   g_w do
5  |  |  mache g_w reflexiv;
6  |  |  if g_w hat alle gewünschten Eigenschaften then
7  |  |  |  return g_w
8  |  |  end
9  |  end
10 end
11 return null;
```

Abbildung 3.5: **Algorithmus**: *Generiere alle nicht-isomorphen Graphen mit bestimmten Eigenschaften. (vereinfacht)*

Um Graphen in Zeile 5 des Algorithmus' reflexiv zu machen, wird von jedem Knoten eine Kante zu sich selbst hinzugefügt. `geng` generiert Graphen durch obere Dreiecksmatrizen von Adjazenzmatrizen und damit nur ungerichtete Graphen ohne Kanten zu sich selbst. Daher haben die von `watercluster2` generierten Graphen auch keine Knoten mit Kanten zu sich selbst. Somit entsteht hierdurch auch keine Redundanz.

Die in Zeile 6 gewünschten Eigenschaften hängen von der Eingabe ab, also von den Parametern, mit denen der Algorithmus aufgerufen wird. Wenn beispielsweise Kripke-Rahmen der Rahmenklasse `S4` untersucht werden sollen, dann muss an dieser Stelle auf Transitivität der Graphen geprüft werden. Dies geschieht über eine Matrizenmultiplikation der Adjazenzmatrix des Graphen mit sich selbst. Da die Adjazenzmatrix die Erreichbarkeit von Knoten in einem Schritt darstellt, repräsentiert ihr Quadrat die Erreichbarkeit von Knoten in zwei Schritten. Der betrachtete Graph ist genau dann transitiv, wenn an jeder Stelle im Quadrat der Adjazenzmatrix des Graphen nur genau dann ein Wert ungleich Null steht, wenn an der gleichen Stelle in der Adjazenzmatrix selber ein Eintrag ungleich Null steht. Eine andere Eigenschaft, die an dieser Stelle überprüft werden kann, ist der Durchmesser des Graphen. Falls an `GraphGenerator` ein Maximum für den erlaubten Durchmesser übergeben wurde, so werden mit dem All-pairs shortest path Algorithmus [APSP/wiki] die kürzesten Pfade zwischen allen Knoten des Graphen berechnet, wovon der längste gleich dem Durchmesser des Graphen ist.

Schwierigkeiten

Die Klasse `GraphGenerator` macht bei der Ausführung des gesamten Programms leider einige Probleme. Dies hängt damit zusammen, dass das Programm `watercluster2` bei manchen Eingaben allem Anschein nach nicht einwandfrei funktioniert.

Bei der verwendeten Windows-Kompilation des Programmes über [cygwin] führen einige Eingaben (z.B. CU oder @[1]) zu Fehlern. Diese haben zur Folge, dass unter Windows nicht alle nicht-isomorphen Graphen generiert werden.

Ein Problem, das sowohl bei der Linux- als auch bei der Windows-Kompilation entsteht, ist, dass das Programm `watercluster2` bei einigen Folgen von Eingaben (z.B. CE gefolgt von @[1]) eine volle CPU-Auslastung hervorruft und nichts ausgibt. Das führt dazu, dass das hier erstellte Programm vor allem bei der Linux-Kompilation an einigen Stellen anhält und nicht weiterläuft. Dies lässt sich zwar durch Terminieren und Wiederstarten des Programmes umgehen, ist aber trotz allem nicht wünschenswert.

Um diese Probleme zu lösen muss ein anderes Programm als `watercluster2` gefunden oder entwickelt werden, welches die Aufgabe lösen kann, alle nicht-isomorphen gerichteten Varianten eines ungerichteten Graphen zu generieren.

3.7 FrameLabeler

Der Programmteil `FrameLabeler` dient dazu, einen Kripke-Rahmen so mit Variablenbelegungen zu ergänzen, dass die daraus resultierende Kripke-Struktur eine übergebene modallogische Formel erfüllt, falls das möglich ist. Zu diesem Zweck wurde zunächst ein Backtracking-Algorithmus entwickelt, welcher die übergebene Formel durch gezielte Variablenbelegungen auf dem Kripke-Rahmen zu erfüllen versucht. Es hat sich herausgestellt, dass die Rekursionstiefe des Algorithmus' zu hoch war, um in dem in dieser Arbeit erstellten Programm zu funktionieren. Im Folgenden wird erläutert, wie der ursprüngliche Algorithmus funktioniert, warum er hier nicht anwendbar ist und wie der dann verwendete BruteForce-Algorithmus funktioniert.

3.7.1 Backtracking

Die Idee dieses Algorithmus' war es, eine gegebene Formel in einer gegebenen Welt eines gegebenen Kripke-Rahmens durch geschickte Variablenbelegungen des Rahmens zu erfüllen.

[1] CU und @ sind hier Kodierungen von ungerichteten Graphen im in [nauty & Traces] verwendeten *graph6*-Format.

Er sollte den Syntaxbaum der übergebenen Formel in Präordnung (also jeweils zuerst die Wurzel und dann deren Nachfolger von links nach rechts) durchgehen und in Abhängigkeit der Knoten Variablen im Kripke-Rahmen in einzelnen Welten setzen oder verbieten. Die Herausforderung bei diesem Algorithmus war es, Variablen für $\Diamond$ und $\Box$ Knoten des Syntaxbaumes korrekt zu setzen. Grundsätzlich funktionierte der Algorithmus wie in Abbildung 3.6 dargestellt.

Parameter: Syntaxbaumknoten *Node*, KripkeFrame *Frame*, Integer *World*

```
1  if Frame ist null then
2  |  return null;
3  end
4  switch Symbol von Node do
5  |  case ⊕ do
6  |  |  Mache linken Nachfolger von Node wahr und rechten falsch in
   |  |   World;
7  |  |  if es gibt einen KripkeFrame F der das erfüllt then
8  |  |  |  return F;
9  |  |  end
10 |  |  Mache rechten Nachfolger von Node wahr und linken falsch in
   |  |   World;
11 |  |  if es gibt einen KripkeFrame F der das erfüllt then
12 |  |  |  return F;
13 |  |  end
14 |  |  return null;
15 |  end
16 |  case □ do
17 |  |  Mache Nachfolger von Node in allen benachbarten Welten von
   |  |   World wahr;
18 |  |  if es gibt einen KripkeFrame F, der das erfüllt then
19 |  |  |  return F;
20 |  |  end
21 |  |  return null;
22 |  end
```

```
23 |  case ◇ do
24 |  |  Mache Nachfolger von Node in mindestens einer benachbarten Welt
   |  |   von World wahr;
25 |  |  if es gibt einen KripkeFrame F, der das erfüllt then
26 |  |  |  return F;
27 |  |  end
28 |  |  return null;
29 |  end
30 |  case Variable do
31 |  |  if Node ist in World nicht als falsch markiert then
32 |  |  |  Mache Node in World wahr und führe den Algorithmus beim
   |  |  |   nächsten Syntaxbaumknoten in Präordnungsdurchlauf aus;
33 |  |  |  Sei F der dadurch entstandene KripkeFrame;
34 |  |  |  if F ist ungleich null then
35 |  |  |  |  return F;
36 |  |  |  end
37 |  |  end
38 |  |  return null;
39 |  end
40 |  case Konstante do
41 |  |  if Node ist wahr in World then
42 |  |  |  Führe den Algorithmus beim nächsten Syntaxbaumknoten in
   |  |  |   Präordnungsdurchlauf aus;
43 |  |  |  Sei F der dadurch entstandene KripkeFrame;
44 |  |  |  if F ist ungleich null then
45 |  |  |  |  return F;
46 |  |  |  end
47 |  |  end
48 |  end
49 end
```

Abbildung 3.6: **Algorithmus label**: Belege KripkeFrame *Frame* so mit Variablen, dass der Syntaxbaum mit *Node* als Wurzel in Welt *World* erfüllt ist.

Da in diesem Algorithmus auch Teilformeln nicht erfüllt werden mussten, wurde zusätzlich der Algorithmus aus Abbildung 3.7 benötigt.

Parameter: Syntaxbaumknoten *Node*, KripkeFrame *Frame*, Integer *World*

```
1  if Frame ist null then
2  |  return null;
3  end
4  switch Symbol von Node do
5  |  case ⊕ do
6  |  |  Mache linken und rechten Nachfolger von Node wahr in World;
7  |  |  if es gibt einen KripkeFrame F der das erfüllt then
8  |  |  |  return F;
9  |  |  end
10 |  |  Mache linken und rechten Nachfolger von Node falsch in World;
11 |  |  if es gibt einen KripkeFrame F der das erfüllt then
12 |  |  |  return F;
13 |  |  end
14 |  |  return null;
15 |  end
16 |  case □ do
17 |  |  Mache Nachfolger von Node in mindestends einer benachbarten
   |  |   Welt von World falsch;
18 |  |  if es gibt einen KripkeFrame F, der das erfüllt then
19 |  |  |  return F;
20 |  |  end
21 |  |  return null;
22 |  end
23 |  case ◊ do
24 |  |  Mache Nachfolger von Node in allen benachbarten Welten von
   |  |   World falsch;
25 |  |  if es gibt einen KripkeFrame F, der das erfüllt then
26 |  |  |  return F;
27 |  |  end
28 |  |  return null;
29 |  end
```

```
30 |  case Variable do
31 |  |  if Node ist in World nicht gesetzt then
32 |  |  |  Markiere Node in World als falsch und führe den Algorithmus
   |  |  |   beim nächsten Syntaxbaumknoten in Präordnungsdurchlauf
   |  |  |   aus;
33 |  |  |  Sei F der dadurch entstandene KripkeFrame;
34 |  |  |  if F ist ungleich null then
35 |  |  |  |  return F;
36 |  |  |  end
37 |  |  end
38 |  |  return null;
39 |  end
40 |  case Konstante do
41 |  |  if Node ist falsch in World then
42 |  |  |  Führe den Algorithmus beim nächsten Syntaxbaumknoten in
   |  |  |   Präordnungsdurchlauf aus;
43 |  |  |  Sei F der dadurch entstandene KripkeFrame;
44 |  |  |  if F ist ungleich null then
45 |  |  |  |  return F;
46 |  |  |  end
47 |  |  end
48 |  end
49 end
```

Abbildung 3.7: **Algorithmus label**: Belege KripkeFrame *Frame* so mit Variablen, dass der Syntaxbaum mit *Node* als Wurzel in Welt *World* nicht erfüllt ist.

Mit der Formulierung, einen Knoten `wahr` zu machen, wie beispielsweise in Zeile 3 des Algorithmus' aus Abbildung 3.6, ist gemeint, dass der Teilbaum mit dem Knoten als Wurzel in der gegeben Welt des Kripke-Rahmens erfüllt werden soll. Dies wird durch einen rekursiven Aufruf des Algorithmus' erledigt. Analog funktioniert das natürlich für das `falsch`-Machen eines Knotens und dem Algorithmus aus Abbildung 3.7, indem der Teilbaum nicht erfüllt wird.

Um beispielsweise in Zeile 14 eine Teilformel in allen benachbarten Welten `wahr` zu machen, reicht eine einfache Schleife, welche diese Teilformel nacheinander in den einzelnen benachbarten Welten `wahr` macht, leider nicht aus. Das liegt daran, dass beispielsweise die Funktionsweise, mit der die gegebene Teilformel in der

ersten Welt `wahr` gemacht wird, einen Einfluss auf die Möglichkeiten haben kann, sie in den übrigen Welten `wahr` zu machen. Wenn die Teilformel in der ersten Welt auf irgendeine Weise `wahr` gemacht wird, so kann es sein, dass sie in anderen Welten nicht mehr `wahr` gemacht werden kann. In einer Schleife würde sich das Programm nicht merken, wie diese Formel in Welt 1 `wahr` gemacht wurde und so annehmen, dass die Teilformel nicht erfüllbar ist, selbst dann, wenn es eine andere Möglichkeit gäbe, die Teilformel in Welt 1 `wahr` zu machen, mit welcher es für die restlichen Welten vielleicht doch möglich gewesen wäre. Um dieses Problem zu lösen, wurde ein Stack implementiert, welcher speichert, in welchen Welten die Formel noch `wahr` gemacht werden soll. Mit einem Stack lässt sich nun wieder mit dem Backtracking-Verfahren arbeiten. Wenn sich also erst in einer späteren Nachbarwelt herausstellt, dass die Formel nicht mehr `wahr` zu machen ist, so kann das Programm nach dem Backtracking-Schema wieder nach hinten „durchreichen“ und an der letzten Stelle, an der alles funktioniert hat, weitermachen.

Diese Herangehensweise hat zwar das Problem gelöst, welches durch die Verwendung von Schleifen entstanden ist, allerdings erzeugt der implementierte Stack eine starke Erhöhung der Rekursionstiefe, da bei jedem modalen Operator im ungünstigsten Fall der gesamte folgende Teilbaum für jede Teilmenge aller Mengen aus der Potenzmenge der Nachfolgerwelten überprüft werden muss. Die dadurch entstehende Rekursionstiefe ist leider für die Belange dieses Programms zu groß, weshalb auf eine weniger zeiteffiziente, aber dafür platzeffizientere Bruteforce-Methode zurückgegriffen wurde.

3.7.2 Bruteforce

Die Bruteforce-Variante des `FrameLabeler` funktioniert relativ simpel. Es wird für jede mögliche Variablenbelegung des Kripke-Rahmens mit den Variablen des gegebenen Syntaxbaumes ausprobiert, ob der Syntaxbaum in der nun mit Variablen belegten Kripke-Struktur erfüllt ist. Es wird zu diesem Zweck eine Binärzahl erstellt, deren Länge gleich der Anzahl von Welten des Kripke-Rahmens multipliziert mit der Anzahl von Variablen des Syntaxbaumes ist. Außerdem werden die Welten des Kripke-Rahmens und die Variablen des Syntaxbaumes jeweils numeriert. Anschließend wird die Binärzahl von 0 beginnend hochgezählt und bei jedem Inkrement über diese iteriert. Ist der Wert der Binärzahl an Stelle i eine 1, so wird die Variable mit dem Index $(i \mod (Anzahl\ an\ Variablen))$ in der Welt mit dem Index $\lfloor \frac{i}{Anzahl\ an\ Variablen} \rfloor$ gesetzt.

Hat der Kripke-Rahmen drei Welten und der Syntaxbaum zwei Variablen, so sähe eine mit der Binärzahl 010111 entstehende Kripke-Struktur beispielsweise folgendermaßen aus:

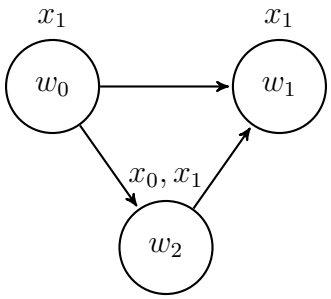

Abbildung 3.8: *Kripke-Struktur, entstanden durch Anwendung des BruteForce-Algorithmus' auf den Kripke-Rahmen* $F = (\{w_0, w_1, w_2\}, \{(w_0, w_1), (w_0, w_2), (w_2, w_1)\})$ *und einen Syntaxbaum mit den Variablen* x_0 *und* x_1 *bei der Binärzahl 010111.*

3.8 FormulaGenerator

Die Klasse `FormulaGenerator` dient dazu, alle erfüllbaren Formeln mit bestimmten Eigenschaften zu generieren. Dies findet statt, indem `TreeGenerator`-Objekte mit gewünschten Parametern erstellt werden, welche Syntaxbäume erzeugen. Diese Syntaxbäume werden dann auf Erfüllbarkeit überprüft und abgespeichert, falls sie erfüllbar sind. Um alle Formeln bis zu einer gegebenen Größe (also bis zu einer bestimmten Anzahl an Knoten) zu generieren, wird der folgende Algorithmus verwendet:

Parameter: Maximale Baumgröße *Size*

```
1  for i = 1,...,Size do
2  |  erstelle TreeGenerator TG mit Parametern maximaler Knotengrad: 3
   |   und maximale Größe: i;
3  |  forall durch TG generierte Bäume T do
4  |  |  if T ist erfüllbar then
5  |  |  |  Definiere MD := modale Tiefe von T;
6  |  |  |  Definiere Formula := Formel, welche T repräsentiert;
7  |  |  |  if sat_md_MD.dat enthält $Formula$ noch nicht then
8  |  |  |  |  Schreibe $Formula$ nach sat_md_MD.dat;
9  |  |  |  end
10 |  |  end
11 |  end
12 end
```

Abbildung 3.9: **Algorithmus**: Generiere alle Formeln bis zu einer übergebenen Größe.

Die $-Symbole, die in Zeile 7 vor und nach der Formel stehen, dienen dazu, zu erkennen, ob eine Formel vollständig in die Datei geschrieben wurde. Das implementierte Programm hat die Funktion, abzubrechen und später weiterzumachen; wenn es also abgebrochen wird, während gerade in die Datei geschrieben wird, dann gibt es in der Datei eine Zeile, welche nicht mit $ beginnt und endet. Da diese Formel aller Wahrscheinlichkeit nach nicht korrekt in die Datei geschrieben wurde, wird vor dem Wiederaufruf des Algorithmus zunächst die `cleanup()`-Methode ausgeführt. Diese überprüft in allen Dateien, in denen Formeln gespeichert sind, ob jede Zeile mit $ beginnt und endet. Ist das in einer Zeile nicht der Fall, wird diese Zeile gelöscht. Nach dem Aufruf von `cleanup()` werden dann zuerst alle übrigen Formeln des letzten benutzen `TreeGenerator` generiert, bevor dann analog zu dem Algorithmus aus Abbildung 3.9 weitergeneriert wird. Um das Weitermachen des Programms zu ermöglichen, wird das `FormulaGenerator`-Objekt alle fünf Sekunden serialisiert, es werden also alle relevanten Informationen des Objektes abgespeichert. Dies wird abwechselnd in zwei Dateien gemacht, da so auch bei einem Programmabbruch während des Serialisierens noch eine valide Datei vorhanden ist.

3.9 FormulaChecker

Die Klasse `FormulaChecker` dient dazu, in allen generierten Formeln nach Formeln der gewünschten Form zu suchen, also nach erfüllbaren Formeln φ und ψ mit $md(\varphi) \neq md(\psi)$ und $\varphi \oplus \psi$ ist unerfüllbar. Dies geschieht folgendermaßen:

```
1  Definiere maxMD := die maximale modale Tiefe, für die schon Formeln
     generiert wurden.;
2  for i = 1,...,maxMD do
3  |  Sei f1 die Datei sat_md_i.dat;
4  |  if f1 existiert then
5  |  |  for j = i+1,...,maxMD do
6  |  |  |  Sei f2 die Datei sat_md_j.dat;
7  |  |  |  if f2 existiert then
8  |  |  |  |  forall Formeln φ aus f1 do
9  |  |  |  |  |  forall Formeln ψ aus f2 do
10 |  |  |  |  |  |  if φ ⊕ ψ ist unerfüllbar then
11 |  |  |  |  |  |  |  Schreibe $φ + ψ$ nach possResults.dat;
12 |  |  |  |  |  |  end
13 |  |  |  |  |  end
14 |  |  |  |  end
15 |  |  |  end
16 |  |  end
17 |  end
18 end
```

Abbildung 3.10: **Algorithmus**: Suche in allen generierten Formeln nach Formeln der gewünschten Form.

Ähnlich wie bei der Klasse `FormulaGenerator` ist auch hier die Funktionalität implementiert, die Überprüfung der Formeln anzuhalten und dort, wo zuletzt angehalten wurde, weiterzumachen. Dazu werden alle relevanten Daten wieder abwechselnd in zwei unterschiedliche Dateien geschrieben, sodass immer zumindest die ältere der beiden Dateien valide Daten enthält.

Die in Zeile 11 in die Datei `possResults.dat` geschriebenen Formeln erfüllen die bisher programmatisch vorausgesetzten Bedingungen. Jedoch sind durch diese Bedingungen nicht unbedingt alle gewünschten Eigenschaften erfüllt. Wegen $md(x \oplus x) = 1$, $x \oplus x \equiv \bot$ und $md(\bot) = 0$ gibt es Formeln, die zwar formal die gegebenen Eigenschaften erfüllen, tatsächlich aber nicht der gewünschten Form

entsprechen.

Um dieses Problem zu lösen könnte man ein Unterprogramm entwickeln, welches nach Überprüfung aller generierten Formeln über alle in `possResults.dat` gespeicherten Formeln der Form $\varphi \oplus \psi$ iteriert, und überprüft, ob die Formeln der gewünschten Form entsprechen. Dies könnte realisiert werden, indem jeweils die Syntaxbäume der Formeln φ und ψ generiert werden und anschließend versucht wird, beide zu minimieren. Diese Minimierung könnte beispielsweise umgesetzt werden, indem zunächst über alle schwach zusammenhängenden, nur aus $\oplus$-Knoten bestehenden Teilgraphen der einzelnen Syntaxbäume iteriert wird. Dabei könnte man jeweils die Vereinigung aller Nachfolger der Knoten eines Teilgraphen bilden, welche keine $\oplus$-Knoten sind. Die durch die Elemente dieser Teilmenge gewurzelten Teilbäume müssten dann paarweise auf Gleichheit überprüft werden. Falls zwei dieser Teilbäume sich gleichen, müssten diese beiden dann aus der Vereinigung entfernt werden und durch einen neuen Knoten mit dem konstanten Wert $\bot$ ersetzt werden. Aufgrund der in Kapitel 3.6 genannten Schwierigkeiten war ich zeitlich leider nicht mehr in der Lage, diesen Algorithmus noch zu implementieren.

Man könnte außerdem noch mehrere bessere Reihenfolgen, in denen Formeln überprüft werden, implementieren. Im zeitlich beschränkten Rahmen dieser Arbeit habe ich dafür bisher nur die Methode `checkAddedDiamonds()` implementieren können, welche bei allen generierten Formeln φ prüft, ob $\varphi \oplus \Diamond\varphi$ unerfüllbar ist. Da die erzeugten Formeln φ erfüllbar sind, sind auch jeweils die Formeln $\Diamond\varphi$ erfüllbar, somit wäre ein unerfüllbares $\varphi \oplus \Diamond\varphi$ ein Ergebnis der gewünschten Form.

3.10 Ausführung und Anwendungshinweise

Anwendung

Das erstellte Programm kann mit unterschiedlichen Parametern aufgerufen werden. Um es über die Konsole zu starten, muss man sich im Ordner `bin` befinden und das Programm dort folgendermaßen aufrufen:

```
java io.IO <Parameter>
```

Ein Aufruf des Programms mit dem ersten Parameter `gen` generiert Formeln, welche dann im Ordner `serlializationFiles` gespeichert werden. Zusätzlich müssen hier noch die maximale Größe der zu generierenden Formeln, die maximale Größe der Kripke-Rahmen, auf denen die Erfüllbarkeit der Formel überprüft werden soll, und die Rahmenklasse (`T` oder `S4`), in der die generierten Formeln erfüllbar sein sollen, angegeben werden. Ein Beispiel für die Parameter eines Aufrufs für maximale Formelgröße 5, maximale Rahmengröße 6 und die Rahmenklasse `T` wäre:

```
gen 5 6 t
```

Wenn in schon generierten Formeln nach Formeln der gewünschten Form gesucht werden soll, dann muss das Programm mit dem ersten Parameter `check` aufgerufen werden. Falls Formeln, die unter den gegebenen Parametern die gewünschte Form erfüllen, gefunden werden, so werden sie im Ordner `serializationFiles` in der Datei `possResults.dat` gespeichert. Hier müssen dann als zusätzliche Parameter nur noch die maximale Größe von Kripke-Rahmen, in denen die Erfüllbarkeit überprüft werden soll, und die gewünschte Rahmenklasse übergeben werden. Ein Beispiel mit maximaler Rahmengröße 4 und Rahmenklasse S4 wäre:

```
check 4 s4
```

Sowohl bei der Generierung als auch bei der Überprüfung von Formeln überprüft das Programm zunächst, ob es schon vorher einen Aufruf mit dem angegebenen Zweck (`check` oder `gen`) gab. Ist das der Fall, so wird aus dieser Datei das abgespeicherte Objekt geladen und es wird an der Stelle, an der es aufgehört hat, weitergemacht. Ist dies nicht der Fall, so startet das Programm von vorne mit der Formelgenerierung oder -überprüfung unter den angegebenen Parametern.

Da das Programm zunächst überprüft, ob noch Dateien von bisherigen Aufrufen existieren, ist es erforderlich, wenn das Programm von vorne laufen soll, die Dateien `checkerData0.dat`, `checkerData1.dat`, `FormulaGenerator0.dat` und `FormulaGenerator1.dat` aus dem Ordner `serlializationFiles` zu löschen.

Schnittstellen zu Hilfsprogrammen

Für diese Arbeit wurden Hilfsprogramme aus [nauty & Traces] und das Programm [showg] verwendet. [showg] dient dazu, die in [nauty & Traces] verwendeten Formate zu dekodieren, während [nauty & Traces] für Berechnungen von Graphen und Bäumen benutzt wird.

Die verwendeten Programme aus [nauty & Traces] sind `gentreeg`, `geng` und `watercluster2`.

Um das Programm auf einem bestimmten Betriebssystem benutzen zu können, müssen die für dieses Betriebssystem kompilierten Programme jeweils im Oberordner `src` und `bin` in dem Ordner `nauty` sein. Für 64-Bit Linux sollte der Inhalt von `nauty` also folgendermaßen aussehen:

`geng`, `gentreeg`, `showg_linux64`, `watercluster2`

Im Code muss in der Datei `TreeGenerator.java` in der Methode `decodeSparse6`

bei der Erstellung des `Process`-Objektes `show` in der Zuweisung von `rt.exec("nauty/showg_linux64 -a")` der Teilstring showg_linux64 durch den Namen des jeweils benutzten showg-Programmes ersetzt werden.

4 Ergebnisse

S4 – transitiv und reflexiv

Für Kripke-Rahmen der Klasse S4 wurden schon relativ früh erfüllbare Formeln φ und ψ unterschiedlicher modaler Tiefe mit $\varphi \oplus \psi$ ist unerfüllbar gefunden. Ein einfaches Beispiel dafür sind die Formeln

$$\varphi := \Diamond a \text{ und } \psi := \Diamond\Diamond a$$
$$\text{und damit}$$
$$\varphi \oplus \psi = \Diamond a \oplus \Diamond\Diamond a.$$

Eine Formel $\varphi \oplus \psi$ ist genau dann unerfüllbar, wenn φ und ψ in allen Rahmen äquivalent sind. Mit den Eigenschaften von S4 aus Kapitel 2.3 kann man die Äquivalenz von $\Diamond a$ und $\Diamond\Diamond a$ in S4 wie folgt zeigen.

Wegen

$$F \models \Diamond\Diamond\varphi \to \Diamond\varphi$$

folgt mit $\varphi := a$

$$\Diamond\Diamond a \to \Diamond a.$$

Wegen

$$F \models \varphi \to \Diamond\varphi$$

folgt mit $\varphi := \Diamond a$

$$(\Diamond a) \to \Diamond(\Diamond a).$$

Insgesamt folgt also

$$\Diamond a \leftrightarrow \Diamond\Diamond a$$

und damit, dass $\Diamond a \oplus \Diamond\Diamond a$ unerfüllbar ist.

Wegen $md(\Diamond a) = 1$ und $md(\Diamond\Diamond a) = 2$, ist dies also offenbar eine Formel der gesuchten Form.

T – transitiv

Für die Rahmenklasse T wurde bei den Durchläufen des Programms mit unterschiedlichen Parametern bisher keine Formel der gesuchten Form gefunden.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Programm erstellt, welches systematisch nach erfüllbaren Formeln φ und ψ unterschiedlicher modaler Tiefe aus dem Fragment der Modallogik über $\{\oplus, \top\}$ sucht, welche die Eigenschaft haben, dass $\varphi \oplus \psi$ unerfüllbar ist.

Zu diesem Zweck wurde neben Repräsentationen von Syntaxbäumen und Kripke-Rahmen bzw. Kripke-Strukturen ein Parser implementiert, welcher modallogische Formeln im String-Format in Syntaxbäume konvertiert. Des Weiteren wurde ein Algorithmus entwickelt, welcher mithilfe von Programmen aus [nauty & Traces] alle nicht-isomorphen Graphen mit bestimmten Eigenschaften erzeugt. Anschließend wurde dann ein Algorithmus entwickelt, welcher die erstellten Graphen mit Variablen belegt, um Kripke-Strukturen zu erhalten, in denen gegebene Formeln erfüllt sind. Außerdem wurde ein Algorithmus entwickelt, welcher mithilfe von Programmen aus [nauty & Traces] alle Formeln mit bestimmten Eigenschaften generiert. Mit all diesen Programmteilen wurde dann schließlich ein Algorithmus entwickelt, welcher systematisch die erstellten Formeln überprüft und so nach Formeln der gewünschten Form sucht.

Probleme gab es bei der Erstellung des Programms unter anderem dadurch, dass der Backtracking-Algorithmus zur Variablenbelegung eines Kripke-Rahmens, um eine gegebene Formel zu erfüllen, zu tief rekursiv war und somit in dem hier entwickelten Programm nicht angewendet werden konnte. Aus diesem Grund wurde auf einen simpleren Bruteforce-Algorithmus zu diesem Zweck zurückgegriffen. Ein weiteres Problem trat beim Aufruf des verwendeten Programms `watercluster2` aus [nauty & Traces] auf. Bei bestimmten Eingaben funktionierte dies nicht wie gewünscht. Dieses Problem konnte nur behelfsmäßig gelöst werden, indem das Programm neu gestartet wird, wenn ein solcher Fehler auftritt.

Bisher wurden für die Rahmenklasse S4 Formeln der gewünschten Form gefunden. Ein Beispiel dafür sind $\varphi = \Diamond a$ und $\psi = \Diamond\Diamond a$. Die beiden einzelnen Formeln sind in S4 erfüllbar und haben eine unterschiedliche modale Tiefe. In S4 sind die beiden Formeln außerdem äquivalent, wodurch deren „exklusives Oder“, also $\varphi \oplus \psi$, in S4 unerfüllbar ist.

Für T wurden bisher allerdings keine solchen Formeln gefunden, also ist noch nicht sicher, ob es sie gibt. Wenn es sie gäbe, dann müsste es zwei erfüllbare modallogische Formeln φ und ψ geben, die eine unterschiedliche modale Tiefe haben und deren „exklusives Oder“ unerfüllbar ist. Damit müssten φ und ψ in T äquivalent sein. Ich nehme an, dass solche Formeln für T nicht existieren, bin aber bisher nicht in der Lage, ihre Existenz zu widerlegen.

In Zukunft könnten die in dem hier erstellten Programm noch bestehenden Probleme dadurch gelöst werden, dass ein anderes Programm zur Erstellung von nicht-

isomorphen gerichteten Varianten eines ungerichteten Graphen verwendet wird. Des Weiteren könnte ein Programm entwickelt werden, welches bei allen möglichen Ergebnissen überprüft, ob sie tatsächlich der gewünschten Form entsprechen, indem beispielsweise Teilformeln wie $\varphi \oplus \varphi$ zu $\bot$ minimiert werden.

Das hier erstellte Programm könnte zukünftig (ggf. in erweiterter Form) eingesetzt werden, um Annahmen zur Erfüllbarkeit auf dem modallogischen Fragment über $\{\oplus, \top\}$ genauer zu betrachten und möglicherweise zu bestätigen oder zu widerlegen. Des Weiteren könnte das Programm auch um weitere logische Operatoren erweitert und so auch für weitere Bereiche der Modallogik angewendet werden. Außerdem könnte das Programm in Abhängigkeit der jeweiligen Fragestellung um bessere Reihenfolgen, in denen die generierten Formeln überprüft werden, erweitert werden.

6 Quellenverzeichnis

6.1 Literatur

[Blackburn et al. 2001]] Blackburn,P., de Rijke,M. & Venema,Y.: Modal Logic, Cambridge University Press (2001)

[Hemaspaandra et al. 2010] Hemaspaandra,E., Schnoor,H. & Schnoor,I.: Generalized modal satisfiability, Journal of Computer and System Sciences 76(7) 561-578 (2010)

[Knuth 2004] Knuth,D.E.: The Art of Computer Programming, A Draft of Sections 7.2.1.4-5: Generating all Partitions, Addison-Wesley (2004)

[Ladner 1977] Ladner,R.E.: The Computational Complexity of Provability in Systems of Modal Propositional Logic, Society for Industrial and Applied Mathematics (1977)

[McKay & Piperno 2014] McKay,B.D. & Piperno,A.: Practical Graph Isomorphism, II, Journal of Symbolic Computation, 60, pp. 94-112, http://dx.doi.org/10.1016/j.jsc.2013.09.003 (2014)

[Parchmann 2011] Parchmann,R.: Skript zur Vorlesung Programmiersprachen und Übersetzer, Leibniz Universität Hannover, Institut für Praktische Informatik, Sommersemester 2011

[Vollmer & Kluge 2015] Vollmer,H. & Kluge,T.: Skript zur Vorlesung Logik und formale Systeme, Leibniz Universität Hannover, Institut für Theoretische Informatik, Sommersemester 2015

6.2 Weitere Quellen

[APSP/wiki]	https://de.wikipedia.org/wiki/Algorithmus_von_Floyd_und_Warshall
[cygwin]	www.cygwin.com
[nauty & Traces]	http://pallini.di.uniroma1.it/, aus [McKay & Piperno]
[showg]	http://users.cecs.anu.edu.au/~bdm/data/formats.html