FernUniversität in Hagen

Fernuniversität in Hagen

Fakultät für Mathematik und Informatik
Lehrgebiet Wissensbasierte Systeme

Abschlussarbeit

im Studiengang Praktische Informatik

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

Thema:	Vervollständigung von partiellen Wissensänderungs-Operatoren
Autor:	Marco Stock <stk.mrc@gmail.com> MatNr. 9761934
Version vom:	4. November 2020
Betreuer:	Prof. Dr. Christoph Beierle

Zusammenfassung

Diese Arbeit thematisiert die Implementierung eines Algorithmus, welcher mittels Brute-Force-Methodik einen partiell spezifizierten Änderungsoperator für wiederholte Wissensänderungen vervollständigt. Dabei können Bedingungen für die Vervollständigung des Operators, mit Hilfe einer Abwandlung der Prädikatenlogik erster Stufe, als zusätzliche Eingabe für den Algorithmus formuliert werden.

Die Ausarbeitung ist wie folgt gegliedert. Das Grundlagenkapitel umfasst eine Einführung in die für den Hauptteil notwendigen Themen. Dazu gehören die Prädikatenlogik erster Stufe (PL1), das Gebiet der Wissensänderungen (Belief Change), sowie die zugehörige Erweiterung der Iterated Belief Revision. Anschließend wird die in der Arbeit verwendete Struktur zur Repräsentation von Änderungsoperatoren, sogenannte Änderungsräume, und eine eingeschränkte Version der PL1, die sogenannte first order logic for belief change (FO-BC), eingeführt. Mit Hilfe dieser Signatur werden die Bedingungen für die Vervollständigung im Algorithmus formuliert. Im darauf folgenden Kapitel wird auf die Implementierung des Programmes eingegangen. Dieses vervollständigt einen gegebenen partiell spezifizierten Änderungsoperator mittels Brute-Force-Algorithmus unter Einhaltung von gegebenen FO-BC Bedingungen, falls ein zugehöriger vollständiger Operator existiert. Es folgen Beispiele der Anwendung anhand von Postulaten aus der Literatur. Um das Thema abzuschließen, wird ein Fazit gezogen und es werden Möglichkeiten zur Optimierung und Erweiterung für das implementierte Programm aufgezeigt.

Abstract

This thesis focuses on the implementation of an algorithm that uses the brute-force method to complete a partially specified change operator for repeated knowledge changes. Conditions for the completion of the Operators can be specified as an additional input for the algorithm, using a modification of first order logic.

The thesis is structured as follows. The first chapter includes an introduction to the necessary topics. This includes the first order logic (FOL), the subject of belief change, as well as its related extension called iterated belief change. After this, the structure, which is further used to represent change operators, so-called change spaces, and a restricted version of the FOL, the so-called first order logic for belief change (FO-BC) is introduced. FO-BC is used to specify conditions for the completion done by the algorithm. The following chapter deals with the implementation of a brute-force algorithm, which can complete a given partially specified change operator in compliance with the given FO-BC conditions, if such an operator exists. This includes examples for the completion based on postulates from literature. To finish the thesis, a conclusion is drawn and possibilities for optimization and extension of the implemented algorithm are pointed out.

Danksagung

An dieser Stelle möchte ich mich zuerst bedanken bei Hr. Kai Sauerwald, wissenschaftlicher Mitarbeiter des Lehrgebiets Wissensbasierte Systeme an der Informatikfakultät der Fernuniversität in Hagen. Seine Person hat mich während der Masterarbeit mit konstruktiver Kritik und Impulsen zur Arbeit an diesem Thema stets unterstützt.

Zudem gilt mein großer Dank Prof. Dr. Christoph Beierle, dessen Kurse und Seminare einen Großteil meines Masterstudiums ausgemacht haben, welcher mich durch seine sehr gute Literatur für das Themengebiet der wissensbasierten Systeme begeistert und mir durch ein Empfehlungsschreiben eine Fortsetzung meiner akademischen Laufbahn ermöglicht hat.

Danke außerdem an meine Familie für die Unterstützung in der Doppelbelastung durch Studium und Arbeit, sowie allen Korrekturlesern der Arbeit.

Marco Stock

Schweinfurt, 04.11.2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1 Einleitung

Diese Arbeit beschäftigt sich mit der Vervollständigung von partiell spezifizierten Operatoren für Wissensänderungen. Neben der Darlegung der theoretischen Grundlagen zu der Problemstellung wird die Implementierung einer solchen automatisierten Vervollständigung thematisiert.

Das Themengebiet der „Künstlichen Intelligenz“ macht einen Großteil der heutigen Forschung in der Informatik aus. Zielsetzung ist die Abbildung von menschlicher Intelligenz auf Computersystemen. Doch bis heute ist die Eigenschaft „Intelligenz“ noch nicht einheitlich definiert. Intelligenz wird oft unter anderem durch die Fähigkeit charakterisiert, sinnvolle Schlussfolgerungen aus, häufig auch widersprüchlichen, Informationen zu treffen.
Neben probabilistischen Ansätzen und nichtmonotonen Logiken bietet das Teilgebiet der Überzeugungsänderung (Belief Revision) einen, auf der klassischen Logik basierten, Ansatz zur Modellierung dieser Fähigkeit. Formal kann dabei die Aktualisierung einer Wissensbasis mit zusätzlicher Information, welche zur Wissensbasis nicht zwingend konsistent ist, über einen logischen Operator beschrieben werden. Bei der Erweiterung zur sogenannten „Iterated Belief Revision“ hängt die neue Wissensbasis nicht nur von der ursprünglichen Wissensbasis und der neuen Information ab, sondern zusätzlich von der Historie, wie die Ausgangs-Wissensbasis entstanden ist. Ein Operator für Wissensänderungen kann beliebig definiert werden. Deshalb existieren in der Literatur Vorschläge für sinnvolle Bedingungen, auch Postulate genannt, an die Wissensänderungen, die der Operator beschreibt.
Die Anzahl an möglichen Wissensänderungen, die durch einen solchen Operator beschrieben werden, steigt sehr schnell mit der Anzahl der betrachteten Eigenschaften. Deshalb werden Operatoren oft nur partiell spezifiziert, d.h. es wird nur ein Teil der möglichen Wissensänderungen explizit angegeben. Die restlichen Wissensänderungen sollen weiterhin den formulierten Postulaten genügen. Ein Programm, welches unter Eingabe eines partiell spezifizierten Wissensänderungsoperators und ergänzenden Postulaten den Operator automatisiert vervollständigt, wird im Rahmen dieser Arbeit implementiert. Die Vervollständigung stellt sicher, dass das Ergebnis jeder möglichen, aber nicht explizit angegebenen Wissensänderung die eingegebenen Postulate erfüllt.

2 Grundlagen

In diesem Kapitel wird auf die Grundlagen eingegangen, auf denen die Implementierung im Hauptteil der Arbeit basiert. Nach der Einführung in die Prädikatenlogik erster Stufe wird das Konzept der Wissensänderung inklusive einschlägiger Postulate aus der Literatur vorgestellt.

2.1 Prädikatenlogik erster Stufe (PL1)

Die Arbeit behandelt einen Logik-basierten Ansatz, um Wissen möglichst sinnvoll zu verwalten. Konkret kommt ein Fragment der Prädikatenlogik erster Stufe zum Einsatz. Dieses Kapitel führt allgemein in die Prädikatenlogik erster Stufe ein. Die speziellen Einschränkungen, die als sogenannte FO-BC (first order logic for belief change) in der Arbeit zum Einsatz kommt, finden sich in Kapitel 4.

2.1.1 Logische Systeme

Die Logik bezeichnet die Lehre vom korrekten Schließen [5]. Es wird versucht zu formalisieren, was gültige Schlüsse von ungültigen Schlüssen unterscheidet. Das Gebiet der Logik kommt ursprünglich aus der Philosophie, findet aber im gesamten Bereich der Informatik und darüber hinaus Anwendung. Bei der hier behandelten Prädikatenlogik handelt es sich um eine spezialisierte Logik auf mathematische Sachverhalte [6].
Historische Erwartungen an eine Logik waren die Beschreibung des menschlichen Denkens bzw. der psychologischen Gesetze [7]. Nach heutiger Auffassung ist diese Aufgabe der Logik allerdings nicht zuzuschreiben, da der Mensch nach moderner Interpretation nicht immer logisch korrekt denkt. Logik kann einen Teil des rationalen Denkens darstellen und Transparenz in Argumentationen bringen. Weiterführende Informationen zur Logik aus einer philosophischen Sichtweise finden sich in [8]. Die weitere Arbeit beschränkt sich auf die Anwendung der Logik im Themenkomplex der Informatik.

Für jedes logische System wird zwischen der Syntax und der Semantik unterschieden. Beide werden für eine Logik exakt definiert. Die Syntax beschreibt durch die Festlegung einer Sprache den Aufbau der möglichen Sätze, bzw. Formeln. Die Semantik stellt den Zusammenhang zwischen rein syntaktischen Symbolen und den Objekten bzw. Begriffen aus der zu repräsentierenden Welt her. Dabei ist die logische Folgerung zwischen zwei Repräsentationen W und B, wie in Abbildung 1 gezeigt, darüber definiert, dass die durch B repräsentierte Semantik in der zu repräsentierenden Welt notwendigerweise aus der Semantik von W folgt. Wichtig ist in dem Zusammenhang der verwendete Zusatz „notwendigerweise“ für die Beziehung zwischen W und B in der zu repräsentierenden Welt. Dabei handelt es sich um die Eigenschaft von klassischen Logiken,

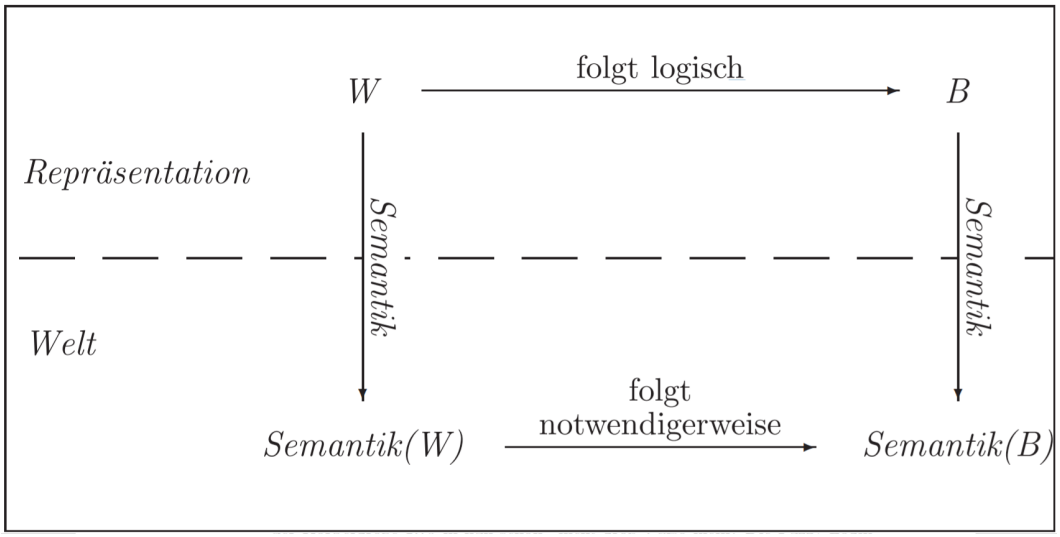

Abbildung 1: Syntax und Semantik einer Logik [1]

dass keine vagen Zusammenhänge modelliert werden können, auch wenn solche in der Realität häufig vorliegen. Auf diese Tatsache wird im Kapitel 2.1.5 auf S.19 nochmal eingegangen.

Die in der Arbeit angewendete Prädikatenlogik erster Stufe (PL1), bzw. im Englischen *first order logic* (FOL), ist die verbreitetste Logik und ist deshalb inzwischen auch sehr gut untersucht. Die im Folgenden vorgestellten Grundlagen zur PL1 sind den Quellen [1], [9], [10] und [11] entnommen.
PL1 ist die Erweiterung einer ebenfalls sehr verbreiteten einfacheren Aussagenlogik, in welcher Formeln ausschließlich aus nullstelligen Prädikatensymbolen (sogenannten Atomen) und Junktoren gebildet werden. Der Ansatz geht unter anderem zurück auf George Boole [12]. In der Aussagenlogik sind allerdings keine verallgemeinerten Relationen darstellbar. Für jede spezifische Relation muss ein entsprechendes Atom eingeführt werden. Durch PL1 kann zwischen Objekten und ihren Eigenschaften unterschieden werden, wodurch verallgemeinerte Beziehungen zwischen Objekten formalisiert werden können. Mit Hilfe dieser Logik wurde versucht menschliche, mathematische und wissenschaftliche Argumentation nachzubilden.
Die Ursprünge gehen zurück auf die Syllogismen von Aristoteles im 4. Jahrhundert vor Christus. Quantifizierte Variablen, welche die Hauptinnovation von PL1 gegenüber der Aussagenlogik darstellen, wurden unabhängig im Jahr 1879 von Gottlob Frege [13] und 1885 von Chraes Saders Peirce vorgeschlagen. Anschließend erweiterten viele Mathematiker, unter anderem Russel [14], Robinson [15], Hilbert [16] und Peano [17] die Logik um wichtige Konzepte. Kurt Gödel wies 1929 in seiner Dissertation [18] die Korrektheit und Vollständigkeit der PL1 nach. Gerhard Gentzen finalisierte in [19] die heute übliche Schreibweise für die PL1-Quantoren ($\forall$, $\exists$). Alfred Tarski [20], [21] beschäftigte sich mit der Semantik der Prädikatenlogik erster Stufe. Den Nachweis, dass PL1 im

Allgemeinen unentscheidbar ist, erbrachte Alonzo Church 1936 [22] mit Hilfe seines λ-Kalküls höherer Ordnung.

Hauptanwendung in der Informatik ist der Bereich des Maschinellen Lernens. Genauer findet die PL1 Anwendung beim Lernen aus strukturierten Daten, beim induktiven logischen Programmieren in wissensbasierten Systemen und beim relationalen Data Mining [10]. Die folgenden Kapitel bilden eine Einführung in die PL1 in der Form, wie sie im Hauptteil dieser Arbeit Anwendung findet.

2.1.2 Syntax

Wie bereits erläutert, beschreibt die Syntax den Aufbau von gültigen Sätzen bzw. Formeln einer Logik. Dazu gehören für die PL1, neben einer Menge V von Variablen, die folgende Menge an Junktoren und Quantoren, welche in Tabelle 1 aufgeführt sind:

Junktoren	$\neg$	Negation („nicht“)
	$\wedge$	Konjunktion („und“)
	$\vee$	Disjunktion („oder“)
	$\Rightarrow$	Implikation („wenn...dann“)
	$\Leftrightarrow$	Äquivalenz („genau dann wenn“)
Quantoren	$\forall$	Allquantor („für alle“)
	$\exists$	Existenzquantor („es gibt“)

Tabelle 1: Junktoren und Quantoren für PL1

Der restliche Teil des Vokabulars wird durch die sogenannte Signatur bereitgestellt. Die Signatur definiert die Bezeichner für die zu betrachtenden Elemente, Funktionen und Aussagen über die Elemente.

Definition 1 (Signatur) *Eine PL1-Signatur* $\Sigma = (Func, Pred)$ *besteht aus:*

- *einer Menge Func von Funktionssymbolen*
- *einer Menge Pred von Prädikatensymbolen*

Die Betonung liegt hier auf der Endung „-symbol“, die Bedeutung der Zeichen wird erst über die Semantik definiert. Jedes Symbol besitzt eine feste Anzahl an Argumenten (Stelligkeit), notiert als *Symbolname/Stelligkeit*. Nullstellige Funktionssymbole, d.h. Funktionssymbole ohne Argumente, werden als *Konstanten* bezeichnet.
Für die Schreibweise von sowohl Funktions- als auch Prädikatensymbolen ist eine Infix-

oder Präfixnotation möglich. Um das Parsing für den zu implementierenden Algorithmus zu vereinfachen, findet in dieser Arbeit überwiegend die Präfixnotation Anwendung.

Mit der eingeführten Unterscheidung in Funktionssymbole und Prädikatensymbole wird anders als in der Aussagenlogik auch zwischen zu bildenden Termen und Formeln unterschieden. Aus dem Vokabular der Signatur werden zusammen mit einer Menge an Variablen Terme gebildet. Terme können wie folgt induktiv definiert werden:

Definition 2 (Terme) *Die Menge $Term_\Sigma(V)$ der Terme über einer Signatur $\Sigma = (Func, Pred)$ und einer Menge V von Variablen ist die kleinste Menge, die die folgenden Elemente gemäß (T1) - (T3) enthält:*

(T1)	x	falls $x \in V$
(T2)	c	falls $c \in Func$ und c hat die Stelligkeit 0
(T3)	$f(t_1, \ldots, t_n)$	falls $f \in Func$ mit der Stelligkeit $n > 0$ und $f(t_1, \ldots, t_n) \in Term_\Sigma(V)$

Durch die Verwendung von Termen zusammen mit den Prädikatsymbolen der jeweiligen PL1-Signatur, Junktoren und Quantoren können Formeln gebildet werden.

Definition 3 (Formeln) *Die Menge $Formel_\Sigma(V)$ der Formeln über einer Signatur $\Sigma = (Func, Pred)$ und einer Menge V von Variablen ist die kleinste Menge, die die folgenden Elemente gemäß (F1) - (F5) enthält:*

(F1)	p	falls $p \in Pred$ und p hat die Stelligkeit 0
(F2)	$p(t_1, \ldots, t_n)$	falls $p \in Pred$ mit der Stelligkeit $n > 0$ und $f(t_1, \ldots, t_n) \in Term_\Sigma(V)$
(F3)	$\neg F$	falls $F \in Formel_\Sigma(V)$
(F4)	$F_1 \wedge F_2$ $F_1 \vee F_2$ $F_1 \Rightarrow F_2$ $F_1 \Leftrightarrow F_2$	falls $F_1, F_2 \in Formel_\Sigma(V)$
(F5)	$\exists x\ F$ $\forall x\ F$	falls $F \in Formel_\Sigma(V)$ und $x \in V$

Die Menge $Formel_\Sigma(V)$ wird in der Literatur gelegentlich auch „Sprache“ genannt, beispielsweise in [9].

Zum Vergleich ist die Aussagenlogik ein Spezialfall mit ausschließlich nullstelligen Prädikatensymbolen. Dadurch können Funktionssymbole und Variablen nicht berücksichtigt werden, was wiederum die Quantoren überflüssig macht. In obiger Definition können also nur Formeln nach (F1), (F3) und (F4) gebildet werden. Dadurch können Beziehung zwischen Objekten nicht beschrieben werden. Formeln beziehen sich dabei allgemein auf alle betrachteten Objekte.
Für bestimmte Formeln und Terme existieren eigene Bezeichnungen. Eine nach (F1) oder (F2) gebildete Formel wird als Atom bezeichnet. Als Literal wird ein Atom inklusive seiner Negierung zusammengefasst. Terme und Formeln ohne Variablen erhalten die Vorsilbe ‚*Grund-* ‘, d.h. Grundterme, Grundformeln bzw. Grundatome.

Die Struktur einer Formel lässt sich über einen Syntaxbaum graphisch veranschaulichen, entsprechend den jeweils festgelegten Bindungsprioritäten.

Definition 4 (Syntaxbaum) *Als Syntaxbaum wird ein endlicher Baum bezeichnet, dessen Knoten mit Formeln beschriftet sind und die folgenden Eigenschaften besitzen:*

- (S1) Die Blätter sind mit Atomen beschriftet, d.h. Formeln der Form (F1) oder (F2)
- (S2) Ist ein Knoten mit einer Formel der Form (F3), d.h. Negation $\neg F$ oder der Form (F5), d.h. Existenzquantor $\exists x\ F$ bzw. Allquantor $\forall x\ F$ beschriftet, dann hat er genau ein Kind, das mit F beschriftet ist
- (S3) Ist ein Knoten mit einer Formel der Form (F4) beschriftet, d.h. Konjunktion $F_1 \wedge F_2$, Disjunktion $F_1 \vee F_2$, Implikation $F_1 \Rightarrow F_2$ oder Äquivalenz $F_1 \Leftrightarrow F_2$, dann hat er genau zwei Kinder und das linke ist mit F_1, das rechte mit F_2 beschriftet.

Die Definition basiert auf der aus [2] entnommenen Definition für einen Syntaxbaum in der Aussagenlogik, erweitert auf Formeln der PL1-Logik. Jede Formel besitzt genau Einen solchen Syntaxbaum. Der Wurzelknoten repräsentiert die gesamte Formel. Diese Eigenschaft wird in der Implementierung im Hauptteil dieser Arbeit zur Formelverarbeitung genutzt. Sinnvollerweise kann die Beschriftung der Knoten auch abgekürzt werden und nur durch den jeweiligen Operator, bzw. bei Quantoren in Kombination mit der zugehörigen Variablen, ersetzt werden. Ein Beispiel für einen so erzeugten PL1 Syntaxbaum für die Formel $(\exists x\ P(x)) \wedge (Q(x,y) \Leftrightarrow Q(y,x))$ stellt Abbildung 2 dar.

Für die in einer Formel auftretenden Variablen wird zwischen freien und gebundenen Variablen unterschieden. Gebunden sind all diejenigen Variablen, über die quantifiziert wird, d.h. entweder über einen Allquantor oder Existenzquantor. Im Syntaxbaum handelt es sich also um Variablen, die im Baum unter einem Quantorknoten, der über die

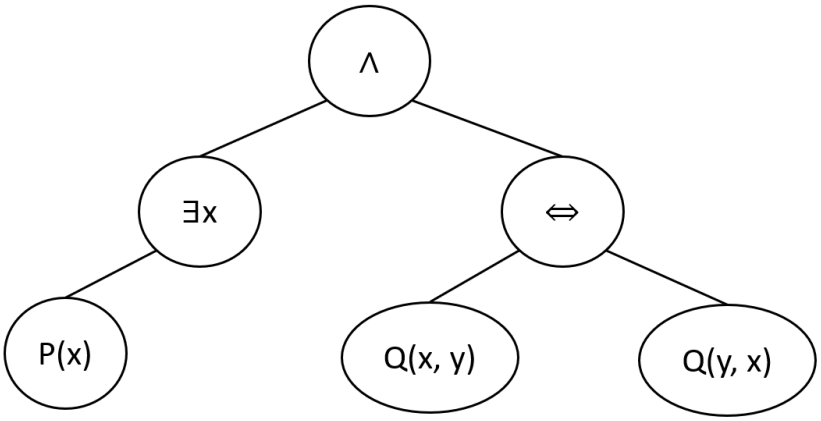

Abbildung 2: Beispiel Syntaxbaum PL1

gleiche Variable quantifiziert, hängen. Freie Variablen sind alle verbleibenden Variablen der Formel. Dabei kann eine Variable sowohl frei als auch gebunden in einer Formel auftreten, wenn die Variable außerhalb des Geltungsbereichs der Quantifizierung nochmal auftritt, d.h. im Syntaxbaum außerhalb des Unterbaums unter dem jeweiligen Quantorknoten. Formeln ohne freie Variablen werden *geschlossen* genannt.

2.1.3 Semantik

Für die Semantik der PL1 wird die Tarskisemantik herangezogen, die entsprechenden Arbeiten dazu wurden im Kapitel 2.1.1 referenziert. In der Tarskisemantik werden allen syntaktischen Symbolen semantische Objekte zugeordnet. Kern der Semantik bildet die *Interpretation* mit der folgenden formalen Definition:

Definition 5 (Interpretation) *Sei $\Sigma = (Func, Pred)$ eine Signatur. Eine Σ-Interpretation $I = (U_I, Func_I, PredI)$ besteht aus:*

- *einer nichtleeren Menge U_I, genannt Trägermenge (oder auch: Universum, Domäne)*
- *einer Menge $Func_I$ von Funktionen*
  $Func_I = \{f_I : \underbrace{U_I \times \cdots \times U_I}_{n\text{-}mal} \to U_I \mid f \in Func \text{ mit der Stelligkeit } n\}$
- *einer Menge $Pred_I$ von Booleschen Funktionen*
  $Pred_I = \{p_I : \underbrace{U_I \times \cdots \times U_I}_{n\text{-}mal} \to BOOL \mid p \in Pred \text{ mit der Stelligkeit } n\}$

Es werden also Zuordnungen, von den verwendeten Symbolen auf das Universum U der Interpretation, getroffen:

Konstante	$\rightarrow$	Element aus U
Funktionssymbol	$\rightarrow$	Funktion über U
Prädikatensymbol	$\rightarrow$	Relation über U
Term	$\rightarrow$	Element aus U
Formel	$\rightarrow$	Wahrheitswert

Die Menge aller Interpretationen $Int(\Sigma)$ zu einer Signatur Σ ist unendlich, da unendlich viele Universen konstruiert werden können. Ebenfalls kann ein Universum eine unendliche Größe besitzen. Ein Beispiel hierfür ist ein Herbrand-Universum über einer Signatur, die mindestens ein einstelliges Funktionssymbol und eine Konstante enthält. Vor der Auswertung einer Formel zu ihrem Wahrheitswert muss zunächst die Variablenbelegung eingeführt werden:

Definition 6 (Variablenbelegung) *Sie $I = (U_I, Func_I, PredI)$ eine Σ-Interpretation und V eine Menge von Variablen. Eine Variablenbelegung (engl. variable assignement) ist eine Funktion*

$$\alpha : V \rightarrow U_I.$$

Für $x \in V$ und $\alpha \in U_I$ bezeichnet $\alpha[x \mapsto a] : V \rightarrow U_I$ die Modifikation von α an der Stelle x zu a. Diese Funktion ist definiert als

$$\alpha[x \mapsto a](y) = \begin{cases} a & \text{falls } x = y \\ \alpha(y) & \text{falls } x \neq y \end{cases}$$

Bei bekannter Variablenbelegung können nun im nächsten Schritt die einzelnen Terme ausgewertet werden.

Definition 7 (Termauswertung) *Gegeben sei ein Term $t \in Term_\Sigma(V)$, eine Σ-Interpretation I und eine Variablenbelegung $\alpha : V \rightarrow U_I$. Die Termauswertung von t unter α, geschrieben $[\![t]\!]_\alpha$, ist gegeben durch eine Funktion*

$$[\![\_]\!]_\alpha : Term_\Sigma(V) \rightarrow U_I$$

und ist definiert durch

$$\begin{aligned} [\![x]\!]_\alpha &= \alpha(x) \\ [\![f(t_1, \ldots, t_n)]\!]_\alpha &= f_I([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha) \end{aligned}$$

Da nun alle Terme zu Elementen des Universums bei gegebener Variablenbelegung ausgewertet werden können, kann im letzten Schritt die Auswertung der Formel für eine

gegebene Variablenbelegung zu Wahrheitswerten erfolgen. Hierbei werden zuerst die atomaren Formeln ausgewertet. Diese erhalten genau dann den Wahrheitswert *TRUE*, wenn die Termauswertung der Argumente in I ein Tupel von Elementen des Universums ergibt, welches in der Relation p_I (siehe Definition der Interpretation) enthalten ist. Anschließend werden die Wahrheitswerte, der einzelnen Auswertungen der atomaren Formeln über die Junktoren und Quantoren, zu einem Wahrheitswert der Gesamtformel verknüpft:

Definition 8 (Wahrheitswert einer Formel unter α) *Gegeben sei eine Formel $F \in Formel_\Sigma(V)$, eine Σ-Interpretation I und eine Variablenbelegung $\alpha : V \to U_I$. Der Wahrheitswert von F unter α, geschrieben $[\![F]\!]_\alpha$, ist gegeben durch eine Funktion*

$$[\![\_]\!]_\alpha : Formel_\Sigma(V) \to BOOL$$

und ist definiert durch

$$[\![P(t_1,\ldots,t_n]\!]_\alpha = \begin{cases} True & \text{falls } ([\![t_1]\!]_\alpha,\ldots,[\![t_n]\!]_\alpha) \in p_I \\ False & \text{sonst} \end{cases}$$

$$[\![\neg F]\!]_\alpha = \begin{cases} True & \text{falls } [\![F]\!]_\alpha = False \\ False & \text{sonst} \end{cases}$$

$$[\![F_1 \wedge F_2]\!]_\alpha = \begin{cases} True & \text{falls } [\![F_1]\!]_\alpha = True \text{ und } [\![F_2]\!]_\alpha = True \\ False & \text{sonst} \end{cases}$$

$$[\![F_1 \vee F_2]\!]_\alpha = \begin{cases} True & \text{falls } [\![F_1]\!]_\alpha = True \text{ oder } [\![F_2]\!]_\alpha = True \\ False & \text{sonst} \end{cases}$$

$$[\![F_1 \Rightarrow F_2]\!]_\alpha = \begin{cases} True & \text{falls } [\![\neg F_1]\!]_\alpha = True \text{ oder } [\![F_2]\!]_\alpha = True \\ False & \text{sonst} \end{cases}$$

$$[\![F_1 \Leftrightarrow F_2]\!]_\alpha = \begin{cases} True & \text{falls } [\![F_1]\!]_\alpha = [\![F_2]\!]_\alpha \\ False & \text{sonst} \end{cases}$$

$$[\![\forall x F]\!]_\alpha = \begin{cases} True & \text{falls für jedes } a \in U_I \text{ gilt: } [\![F]\!]_{\alpha[x\mapsto a]} = True \\ False & \text{sonst} \end{cases}$$

$$[\![\exists x F]\!]_\alpha = \begin{cases} True & \text{falls es ein } a \in U_I \text{ gibt, so dass: } [\![F]\!]_{\alpha[x\mapsto a]} = True \\ False & \text{sonst} \end{cases}$$

Da der Wahrheitswert ganzer Sätze über die Junktoren eindeutig durch die einzelnen Teilsätze bestimmt ist, ist die PL1-Logik *wahrheitsfunktional*. Nach der nun möglichen Bestimmung des Wahrheitswerts einer Formel für eine gegebene Variablenbelegung,

ist die Auswertung des Wahrheitswertes der Formel für sämtliche Variablenbelegungen von Interesse. Für geschlossene Formeln ist der Wahrheitswert unabhängig von der Variablenbelegung α. Deshalb werden für die folgende Definition freie Variablen analog zu allquantifizierten Variablen behandelt, also jeweils der sogenannte *Allabschluss* der Formel betrachtet. Wenn eine Variable nicht durch einen Existenzquantor gebunden ist, wird die Formel insgesamt nur dann zum Wahrheitswert *TRUE* ausgewertet, wenn die Auswertung der Formel unter allen Belegungen dieser Variablen, ebenfalls dem Wahrheitswert *TRUE* entspricht.

Definition 9 (Wahrheitswert einer Formel) *Für eine Formel $F \in Formel_\Sigma(V)$ und eine Σ-Interpretation I ist der Wahrheitswert von F in I gegeben durch die Funktion*

$$[\![F]\!]_I : Formel_\Sigma(V) \to BOOL$$

und ist definiert durch

$$[\![F]\!]_I \quad = \quad \begin{cases} True & \begin{cases} \text{falls } [\![F]\!]_\alpha = True \\ \text{für jede Variablenbelegung } \alpha : V \to U_I \end{cases} \\ False & \text{sonst} \end{cases}$$

Für eine Interpretation I, für die der Wahrheitswert von F unter I gemäß obiger Definition *TRUE* beträgt, wird der Begriff des *Modells* eingeführt. Eine Interpretation I ist Modell einer Formel F, geschrieben $I \models F$, gdw. $[\![F]\!]_I = true$. Das beschreibt die Erfüllungsrelation $\models_\Sigma \subseteq Int(\Sigma) \times Formel_\Sigma(V)$ zwischen Formeln und Interpretationen der PL1-Logik. Alternativ wird die Formulierung „I erfüllt F“ verwendet. Der Begriff kann auch auf Formelmengen erweitert werden. Eine Interpretation ist Modell einer Formelmenge FM, gdw. es gleichzeitig Modell jeder Formel in der Formelmenge $F \in FM$ ist.
Eine Formel, bzw. Formelmenge, kann durch keine, eine begrenzte Anzahl, oder alle möglichen Interpretationen erfüllt werden. Entsprechend der Erfüllbarkeit werden PL1-Formeln in vier Klassen eingeteilt:

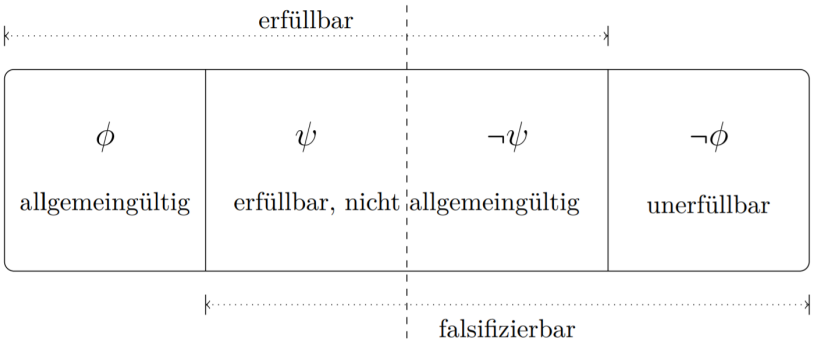

Abbildung 3: PL1 Formelklassifizierung [2]

Definition 10 (Erfüllbarkeit von PL1-Formeln) *Eine Formel, bzw. Formelmenge, F ist*

- *erfüllbar (konsistent) gdw. $Mod_\Sigma(F) \neq \emptyset$, d.h. wenn sie von wenigstens einer Interpretation erfüllt wird.*
- *unerfüllbar (widersprüchlich, inkonsistent) gdw. $Mod_\Sigma(F) = \emptyset$, d.h. wenn sie von keiner Interpretation erfüllt wird.*
- *allgemeingültig (Tautologie) gdw. $Mod_\Sigma(F) = Int(\Sigma)$, d.h. wenn sie von jeder Interpretation erfüllt wird.*
- *falsifizierbar gdw. $Mod_\Sigma(F) \neq Int(\Sigma)$, d.h. wenn sie von wenigstens einer Interpretation nicht erfüllt wird, also falsifiziert.*

Die Beziehungen zwischen den Klassen gehen aus Abbildung 3 hervor.

2.1.4 Deduktion

Mit Hilfe von PL1-Formeln kann in einem wissensbasierten System das Wissen repräsentiert werden. Für die Implementierung von „künstlicher Intelligenz“ ist zusätzlich eine Deduktionskomponente notwendig, die Schlussfolgerungen aus dem gegebenen Wissen über logische Folgerungen ableiten kann. Ein Beispiel für ein wissensbasiertes System mit PL1-Komponenten, in Form einer automatischen Fehlerdiagnose, findet sich in [23]. Das *Deduktionstheorem* beschreibt die klassisch logische Folgerung in der PL1-Logik. Für die einzuführende binäre Relation wird dasselbe Zeichen $\models_\Sigma$ wie für die Erfüllungsrelation verwendet:

Definition 11 (Logische Folgerung) *Die binäre Relation*

$$\models_\Sigma \subseteq 2^{Formel_\Sigma(V)} \times 2^{Formel_\Sigma(V)}$$

ist definiert durch

$$F \models_\Sigma G \quad \text{gdw.} \quad Mod_\Sigma(F) \subseteq Mod_\Sigma(G)$$

Dabei können F und G sowohl einzelne Formeln als auch Formelmengen sein. $F \models_\Sigma G$ bedeutet, dass G logisch, bzw. semantisch, aus F folgt. Das ist ein zentraler Begriff der Prädikatenlogik. Logische bzw. semantische Äquivalenz zwischen Formel (-mengen) liegt vor, wenn die Menge der Modelle beider exakt gleich ist. Semantische Äquivalenzen für PL1-Formeln sind in der Literatur zu finden, z.B. in [9, p. 32]. Diese werden unter anderem zum Umformen von Formeln verwendet. Ziel ist meist eine Normalform, auf der anschließend die Deduktion stattfindet. Hier sind vor allem die konjunktive (KNF) und disjunktive (DNF) Normalform zu nennen. Bei der oft verwendeten Klauselform handelt es sich um eine Mengendarstellung der KNF.

Für die Aussagenlogik ist die Fragestellung, ob eine Formel logisch aus einer Anderen folgt vergleichsweise leicht zu beantworten. Hierfür können mit maximal 2^n Versuchen, wobei n die Anzahl an verschiedenen Atomen der Formeln ist, alle Modelle der einen Formel ermittelt und für die Andere überprüft werden. Möglich ist das z.B. mit Hilfe einer Wahrheitstafel. Das Problem ist für die Aussagenlogik also entscheidbar.
Für die PL1 ist die Anzahl an möglichen Universen im Allgemeinen unendlich, siehe Kapitel 2.1.3, wodurch es auch eine unendliche Anzahl an Interpretationen gibt. Eine Überprüfung aller Interpretationen ist also in finiter Zeit nicht möglich. Das Problem der logischen Folgerung ist für die PL1-Logik damit unentscheidbar. Es ist jedoch semi-entscheidbar, d.h. es kann algorithmisch in finiter Zeit bestätigt werden, dass eine logische Folgerung $F \models G$ zutrifft. Allerdings würde der Algorithmus für eine Eingabe, die keine korrekte logische Folgerung darstellt, nicht zwingend terminieren. Es existieren in der Literatur für PL1 inzwischen eine große Anzahl an Kalkülen, welche die Basis für einen solchen Algorithmus bilden können. Ein bekanntes Beispiel ist das Resolutionskalkül nach Robinson [15] von 1965, welches die logische Folgerung $F \models G$ bestätigt, indem die Unerfüllbarkeit von $F \wedge \neg G$ gezeigt wird. Das negative Testkalkül ist widerlegungsvollständig und besteht nur aus einem einzigen Axiom und einer Inferenzregel. Falls die Konjunktion $F \wedge \neg G$ tatsächlich unerfüllbar ist, kann die Unerfüllbarkeit, bzw. damit die Bestätigung der logischen Folgerung $F \models G$, in endlicher Zeit abgeleitet werden. Die logische Programmiersprache Prolog benutzt die Resolution auf vereinfachten Klauseldarstellungen, den nach Alfred Horn benannten Hornklauseln. Peirce unterscheidet in [24] die Inferenz in Deduktion, Abduktion und Induktion. Aussagen, die per Deduktion abgeleitet werden, sind stets korrekt. Bei der Abduktion und

Induktion sind die getroffenen Schlussfolgerungen nicht zwingend korrekt, da hier regelhafte Beobachtungen bzw. einzelne Sachverhalte zu Grunde liegen. Zu diesen Arten von Inferenz gibt es auch Umsetzungen auf Basis der PL1-Logik, siehe dazu z.B. [25] für eine mögliche Umsetzung von Abduktion. Abduktion ist besonders wegen der Unentscheidbarkeit der logischen Folgerung problematisch. Da Deduktion bzw. Inferenz im Allgemeinen keinen zentralen Inhalt der weiteren Arbeit darstellen, wird an dieser Stelle für weitere Ausführungen auf die bereits angegebene Literatur [1], [9], [10] und [11] verwiesen.

2.1.5 Erweiterungen

Da die Prädikatenlogik erster Stufe, wie bereits erwähnt, eine der am besten untersuchtesten Logiken ist, existieren bereits einige Vorschläge für Erweiterungen. Mit Hilfe der Prädikatenlogik bestand die Hoffnung mathematische Beweise zu automatisieren. Bis auf wenige Ausnahmen ist das in dem erwarteten Maß allerdings nicht gelungen. Genauso wenig wie eine Axiomisierung des gesamten Wissens der Welt mit Hilfe der Logik möglich ist, woraus allgemeingültige Schlüsse gezogen werden könnten. Der Autor in [26] kritisiert die PL1-Logik auf der einen Seite als zu ausdrucksstark, was sich beispielsweise durch die Unentscheidbarkeit der Allgemeingültigkeit und logischen Folgerung äußert. Auf der anderen Seite als zu ausdrucksschwach. Die PL1-Logik kann demnach kein vages, unvollständiges und zeitabhängiges Wissen darstellen. Auch wenn Solches die Realität oft besser wiedergeben könnte. Vorschläge, um Unsicherheit in die PL1 zu integrieren, sind in [27], [28] und [29] zu finden. Für die Verarbeitung von zeitabhängigem Wissen bietet sich die Temporallogik an, eine Form der Modallogik. Dazu sei auf die Veröffentlichungen [30], [31] und [32] verwiesen.
Weitere mögliche Erweiterungen aus der Literatur stellen die Einführung einer Identitätsrelation (*PL1 mit Gleichheit*), wie in [33], oder die Erweiterung der Existenzquantoren zu sogenannten Anzahlquantoren in [34] dar. Hierüber kann nicht nur die Existenz eines Elements gefordert werden, sondern konkreter auch die Existenz einer genauen Anzahl k an Elementen. Der Autor in [35] beschäftigt sich mit Einbindung von partiell definierten Prädikaten durch die Einführung eines dritten Wahrheitswertes.
Eine Alternative zur PL1 stellen Logiken höherer Ordnung dar. Diese können für die Verarbeitung durch Rechner in PL1 Sätze überführt werden [11]. Gängig sind hier nach [36] z.B. Lambda-Logik, Binding-Logik, Monadic Second-Order (MSO) und die Prädikatenlogik zweiter Stufe. Bei der MSO werden Variablen eingeführt, die Sets von Elementen beschreiben können. Bei der Prädikatenlogik zweiter Stufe werden Variablen für Beziehungen zwischen Objekten eingesetzt. Allerdings ist die Prädikatenlogik zweiter Stufe im Gegensatz zur PL1-Logik nicht vollständig. Das resultiert darin, dass sich die Logik als Basis für Deduktion mit Standard Semantik im Vergleich zu PL1 nicht vergleichbar gut eignet [37, p. 46].

2.2 Wissensänderung (Belief Change)

Für die Modellierung von unsicherem logischen Schließen existieren drei verschiedene bekannte Ansätze [38]. In der nicht-monotonen Logik werden PL1 Fakten durch sogenannte „Defaults“ erweitert, mit denen mögliche Extensionen der Faktenmenge generiert werden können [39]. In probabilistischen Ansätzen wird dagegen eine Wahrscheinlichkeitsfunktion zu Grunde gelegt und mit Hilfe des Satz von Bayes für neue Informationen adaptiert [40][41]. Die dritte Art der Modellierung unsicheren Wissens und die, mit der sich diese Ausarbeitung im Folgenden detaillierter beschäftigt, ist die Überzeugungsänderung; im Englischen: „Belief Change“. Dabei wird das Wissen eines Agenten bzw. seiner Welt formalisiert als sein Belief State. Mit neuen Informationen ändert sich sein Wissen bzw. seine Weltanschauung auf eine bestimmte Art und Weise, die mit Hilfe eines Revisionsoperators formalisiert wird.
Die Art der Modellierung entspricht in der Psychologie dem Vorgang der Meinungsänderung. Dazu gehören speziell die Bereiche der Vorurteile, Argumentation und Überzeugung [42]. In der Informatik findet diese Modellierung Anwendung bei Diagnosesystemen, in der abduktiven Argumentation und bei Aktualisierung von Datenbanken [43].

Die Überzeugungsänderung kann auf verschiedene Arten durchgeführt werden. Grundsätzlich sind dabei drei Prinzipien zu unterscheiden, anhand derer die formalisierte Änderung des Wissens erfolgen kann (wörtlich aus [42]):

- **Konservatismus**: Überzeugungen werden beibehalten, solange kein spezieller Grund gegen sie spricht. Dies ist besonders bei Überzeugungen der Fall, die man irgendwann angenommen hat und seitdem für begründet hält, von denen man aber nicht mehr sagen kann, wie sie zustande gekommen sind.

- **Epistemische Verwurzelung**: Bei der Revision werden nur die Überzeugungen aufgegeben, die so wenig grundlegend wie möglich sind. Damit ist gemeint, dass Teile unseres Wissens und unserer Überzeugungen fester verankert und besser begründet sind als andere. Bei widersprüchlichen Informationen sollten diese besser verankerten Überzeugungen also weniger leicht aufgegeben werden. Beispiele dafür sind z.B. Naturgesetze, wie die Schwerkraft oder Wissenshierarchien, wie die Tatsache, dass Pinguine Vögel sind und Vögel zu den Lebewesen gehören. Das Merkmal „fliegen können“ ist in diesem Fall epistemisch weniger verwurzelt.

- **Minimale Änderung**: Es wird diejenige Überzeugung revidiert, die zu einer minimalen Anzahl von Nebeneffekten auf andere Überzeugungen führt.
  Hierzu zählt auch das AGM-Modell, auf welches im nächsten Kapitel detaillierter eingegangen wird. Das AGM–Modell ist eine formale Repräsentation für Überzeugungsänderung mit minimaler Änderung und in der zugehörigen Veröffentlichung [44] durch Postulate definiert.

2.3 AGM-Postulate

Das AGM-Modell betrachtet drei Typen von Überzeugungsänderungen [4] [45, 351 ff]:

- **Expansion** $K + p = Cn(K \cup \{p\})$: Das Ergebnis der Expansion ist das kleinste deduktiv abgeschlossene Belief Set, das sowohl K als auch p enthält. Das aktuelle Wissen wird um eine neue (konsistente) Information erweitert.
- **Kontraktion** $K \div p$: Bei der Kontraktion wird p aus K entfernt, d.h. p darf im neuen Belief Set nicht mehr ableitbar sein. Das Ergebnis bildet ein Subset von K, aus welchem allerdings keine Elemente unnötigerweise entfernt wurden.
- **Revision** $K * p$: Bei der Revision wird p zu K hinzugefügt, aber gleichzeitig werden andere Formeln wieder entfernt, damit das resultierende Set konsistent ist.

AGM-Kontraktion und Revision sind eng miteinander verbunden. So können mit Hilfe der Levi und Harper Identität Revisionsoperatoren in Kontraktionsoperatoren und vice-versa überführt werden.

Definition 12 (Levi und Harper Identität auf Belief States [46]) *Levi und Harper Identität auf deduktiv abgeschlossenen Belief Sets sind wie folgt definiert:*

$$\begin{aligned} \textbf{(Levi Identität)} \quad & K * p = (K \div \neg p) + p \\ \textbf{(Harper Identität)} \quad & K \div p = K \cap (K * \neg p) \end{aligned}$$

Ziel bei Kontraktion und Revision ist den Informationsverlust so gering wie möglich zu halten. Das „Entfernen“ von Elementen bei der Kontraktion klingt trivial. Das folgende Beispiel zeigt aber, dass es mehrere Möglichkeiten gibt die Operation durchzuführen und die Entscheidung, welche Möglichkeit die „bessere“ Lösung darstellt, ist ggf. schwierig zu treffen.

$$K = Cn(a, a \Rightarrow b)$$

$$K \div b = Cn(a) \qquad \textit{oder} \qquad K \div b = Cn(a \Rightarrow b) \qquad ?$$

Eine mögliche Interpretation des Beispiels wäre: a = Es hat geregnet; b = die Straße ist nass. Der aktuelle Wissensstand ist, dass es geregnet hat und man weiß, dass wenn es geregnet hat, die Straße nass ist. Aus den beiden Aussagen kann also ebenfalls die Überzeugung, dass die Straße nass sein muss, abgeleitet werden. Durch einen Blick

aus dem Fenster wird nun die neue Information erhalten, dass die Straße allerdings doch nicht nass ist. Dadurch stellt sich die Frage, an welcher Überzeugung festgehalten werden soll und welche Überzeugung verworfen wird, um eine aktualisierte konsistente Wissensbasis zu erhalten. Eine Möglichkeit stellt das Verwerfen der Überzeugung, dass es geregnet hat, dar. Alternativ könnte auch das regelhafte Wissen verworfen werden, dass die Straße nass wird, wenn es regnet.

Eine grundlegende Veröffentlichung in dem Bereich bildet das Paper der AGM-Theorie (Anfangsbuchstaben der drei Autoren) aus dem Jahr 1985, welches unter [44] zu finden ist. Hier wurde die Partial Meet Kontraktion eingeführt, bei der nicht nur eine Lösung ausgewählt wird, sondern die Schnittmenge aus den vielversprechendsten Lösungen gebildet wird.

2.3.1 Formale Voraussetzungen

Im AGM-Modell wird die Überzeugung eines Agenten über ein deduktiv abgeschlossenes Set von Formeln ausgedrückt. Deduktiv abgeschlossene Formelmengen $K = Cn(K)$ werden auch als *Belief Sets* bezeichnet. Interpretationen werden im Kontext von Wissensänderungen auch als Welten bezeichnet. Die Menge aller betrachteten Interpretationen $\omega_0, \omega_1, \ldots$ einer Sprache L wird im Folgenden als Menge aller *Welten* Ω bezeichnet. Die Anzahl der Welten wird mit n ausgedrückt: $\omega_0, \omega_1, \ldots, \omega_n \in \Omega$.

Definition 13 (Quasiordnung) *Eine Quasiordnung $\leq$ (englisch preorder) bezeichnet eine zugleich reflexive und transitive Ordnung.*

$$\omega_1 < \omega_2 \textit{ wird dabei definiert als } \omega_1 \leq \omega_2 \wedge \omega_1 \not\leq \omega_2.$$

$$\omega_1 = \omega_2 \textit{ wird dabei definiert als } \omega_1 \leq \omega_2 \wedge \omega_1 \leq \omega_2.$$

Definition 14 (Totalität) *Eine Quasiordnung $\leq$ wird als total bezeichnet (englisch total pre-order), wenn alle Elemente der betrachteten Menge Ω bezüglich der Ordnung vergleichbar sind, d.h. für zwei Elemente $\omega_1, \omega_2 \in \Omega$:*

$$\omega_1 \leq \omega_2 \quad \vee \quad \omega_2 \leq \omega_1$$

2.3.2 AGM-Kontraktion

AGM-Revision und Kontraktion sind in [44] durch Postulate charakterisiert. Die AGM-Postulate für die Kontraktion sind in Tabelle 2 zusammengefasst. Die ersten sechs Postulate definieren die *partial meet contraction*. Die letzten beiden zusätzlichen Postulate ergänzen die Definition für die *transitively relational partial meet contraction*. Über die Levi und Harper Identität lässt sich erklären, dass für die Revision ähnliche Postulate gelten.

(G1)	Closure	$K \div p = Cn(K \div p)$
(G2)	Success	*if* $p \notin Cn(\varnothing)$ *then* $p \notin Cn(K \div p)$
(G3)	Inclusion	$K \div p \subseteq K$
(G4)	Vacuity	*if* $p \notin Cn(K)$ *then* $K \div p = K$
(G5)	Extensionality	*if* $(p \Leftrightarrow q) \in Cn(\varnothing)$ *then* $K \div p = K \div q$
(G6)	Recovery	$K \subseteq (K \div p) + p$
(G7)	Conjunctive inclusion	*if* $p \notin K \div (p \wedge q)$ *then* $K \div (p \wedge q) \subseteq K \div p$
(G8)	Conjunctive overlap	$(K \div p) \cap (K \div q) \subseteq K \div (p \wedge q)$

Tabelle 2: AGM-Postulate Kontraktion [4]

2.3.3 AGM-Revision

Für die *partial meet revision* sind sechs analoge Postulate zu denen der Kontraktion definiert, siehe Tabelle 3. Recovery wird ersetzt durch die Forderung nach Konsistenz. Alle acht Postulate zusammen formalisieren auch hier die sogenannte *transitively relational partial meet revision*.

(G*1)	Closure	$K * p = Cn(K * p)$
(G*2)	Success	$p \in K * p$
(G*3)	Inclusion	$K * p \subseteq K + p$
(G*4)	Vacuity	*if* $\neg p \notin K$ *then* $K * p = K + p$
(G*5)	Extensionality	*if* $(p \Leftrightarrow q) \in Cn(\varnothing)$ *then* $K * p = K * q$
(G*6)	Consistency	$K * p$ *is consistent if* p *is consistent*
(G*7)	Superexpansion	$K * (p \wedge q) \subseteq (K * p) + q$
(G*8)	Subexpansion	*if* $\neg q \notin Cn(K * p)$ *then* $(K * p) + q \subseteq K * (p \wedge q)$

Tabelle 3: AGM-Postulate Revision [4]

2.3.4 Übertragung der Postulate auf die Aussagenlogik

Die AGM-Postulate sind für deduktiv abgeschlossene Belief Sets formuliert. Gerade für die Implementierung einer Belief Revision oder Kontraktion mit Hilfe der Aussagenlogik ist das ungeeignet.

Übertragung der Revisionspostulate Die Autoren in [43] stellt eine Korrespondenz zwischen aussagenlogischen Formeln und Belief Sets her, wodurch sich auch eine Korrespondenz zwischen der Revision auf Belief Sets $K * \mu$ und auf aussagenlogischen Formeln $\psi \circ \mu$ ergibt.
Für jede aussagenlogische Wissensbasis ψ existiert ein korrespondierendes deduktiv abgeschlossenes Belief Set $K = \{\phi \mid \psi \vdash \phi\}$. Andersherum kann ein Belief Set durch mehrere Wissensbasen beschrieben werden. Auf der Basis hat der Autor die allgemeinen AGM-Revisionspostulate (G*1) - (G*8) in sechs Bedingungen (R1) - (R6) für die Definition eines Revisionsoperators auf der Aussagenlogik überführt:

(R1) $\psi \circ \mu$ *implies* μ
(R2) *if* $\psi \wedge \mu$ *is satisfiable, then* $\psi \circ \mu \equiv \psi \wedge \mu$
(R3) *if* μ *is satisfiable, then* $\psi \circ \mu$ *is also satisfiable*
(R4) *if* $\psi_1 \equiv \psi_2$ *and* $\mu_1 \equiv \mu_2$ *then* $\psi_1 \circ \mu_1 \equiv \psi_2 \circ \mu_2$

Proposition 1 (Überführung der AGM-Revisionspostulate 1 [43]) *Ein Revisionsoperator* $\circ$ *erfüllt die Bedingungen (R1) - (R4), gdw. sein zugehöriger Revisionsoperator auf deduktiv abgeschlossenen Belief Sets* $*$ *die sechs AGM-Postulate (G*1) - (G*6) der partial meet revision erfüllt.*

- (R1) entspricht dabei der Forderung, dass das neue Wissen in der aktualisierten Wissensbasis aufgenommen werden muss.
- (R2) fordert, dass der offensichtliche Weg befolgt wird. Wenn es zu keiner Inkonsistenz führt, wird die Wissensbasis lediglich um das neue Wissen erweitert.
- (R3) verhindert, dass durch die Revision eine Inkonsistenz eingeführt wird.
- (R4) beschreibt die Irrelevanz der Syntax.

Auch die beiden ergänzenden AGM-Postulate für eine *transitively relational partial meet revision* (G*7) und (G*8) überführt der Autor in die Forderungen (R5) und (R6) für einen Revisionsoperator auf der Aussagenlogik:

(R5) $(\psi \circ \mu) \wedge \phi$ *implies* $\psi \circ (\mu \wedge \phi)$
(R6) *if* $(\psi \circ \mu) \wedge \phi$ *is satisfiable, then* $\psi \circ (\mu \wedge \phi)$ *implies* $(\psi \circ \mu) \wedge \phi$

Diese Forderungen entsprechend dem Anspruch, dass die Revision mit „minimaler Änderung“ erfolgen soll. Dafür wird eine Metrik eingeführt, mit der der Abstand zwischen den Modellen gemessen wird. Es wird das Modell von μ bevorzugt, das zu der Menge der Modelle von ψ den geringsten Abstand aufweist.

- (R5) besagt Folgendes: Wenn eine Interpretation in der Menge der Modelle von μ die geringste Distanz zu den Modellen von ψ aufweist und die Interpretation auch in der Teilmenge der Modelle von ψ und μ enthalten ist, dann ist die Interpretation auch dort die Interpretation mit der geringsten Distanz zu den Modellen von ψ.
- (R6) verhindert Folgendes: Ohne diese Bedingung könnte eine Interpretation I für $\psi \circ (\mu \wedge \phi)$ die geringste Distanz aufweisen und gleichzeitig für $(\psi \circ \mu) \wedge \phi$ kein Modell sein. Dann hätte das Modell J von $(\psi \circ \mu) \wedge \phi$ (das Modell muss existieren, siehe Voraussetzung R6) allerdings die geringste Distanz zu den Modellen von ψ aus der Menge der Modelle von μ und das Modell I eine geringere Distanz als J für $\psi \circ (\mu \wedge \phi)$. Sobald ein Modell für $(\psi \circ \mu) \wedge \phi$ existiert, muss deshalb das Modell mit der geringsten Distanz gleichzeitig auch das mit der geringsten Distanz für $\psi \circ (\mu \wedge \phi)$ sein.

Proposition 2 (Überführung der AGM-Revisionspostulate 2 [43]) *Ein Revisionsoperator* $\circ$ *erfüllt die Bedingungen (R1) - (R6), gdw. sein zugehöriger Revisionsoperator auf deduktiv abgeschlossenen Belief Sets* $*$ *die acht AGM-Postulate (G*1) - (G*8) der transitively relational partial meet revision erfüllt.*

Definition 15 (faithful assignment) *Eine Funktion, die jede mögliche Formel* ψ *in einer aussagenlogischen Sprache L einer totalen Quasiordnung* $\leq_\psi$ *auf allen Welten* Ω *zuordnet, wird als faithful assignment bezeichnet, gdw. folgende drei Bedingungen zutreffen:*

(1) $\omega_1, \omega_2 \models \psi \;\Rightarrow\; \omega_1 =_\psi \omega_2$
(2) $\omega_1 \models \psi$ *and* $\omega_2 \not\models \psi \;\Rightarrow\; \omega_1 <_\psi \omega_2$
(3) $\psi \equiv \phi \;\Rightarrow \leq_\psi = \leq \phi$

Diese Bedingungen fordern, dass ein Modell nicht echt kleiner als ein anderes Modell sein kann und ein Modell echt kleiner sein muss als jede Interpretation, die kein Modell

ist. Mit Hilfe eines Repräsentationstheorems führen die Autoren damit die Beziehung zwischen den Postulaten (R1) - (R6) und einem Revisionsmechanismus, der auf totalen Quasiordnungen basiert, ein:

Proposition 3 (Repräsentationtheorem Revision [43]) *Wenn eine totale Quasiordnung $\leq_\psi$ für Interpretationen existiert und die Zuordnung zu jeder Wissensbasis den Bedingungen für ein faithful assignment genügt, so dass $Mod(\psi \circ \mu) = Min(Mod(\mu), \leq_\psi)$ gilt, erfüllt der dadurch definierte Revisionsoperator die AGM-Postulate für die transitively relational partial meet revision, d.h. die AGM-Revisionspostulate (G*1) - (G*8).*

$Min(Mod(\mu), \leq_{psi})$ meint die kleinsten Interpretationen unter der angegebenen Ordnungsrelation von der angegebenen Menge von Interpretationen (hier $Mod(\mu)$).

Übertragung der Kontraktionspostulate Analog zur Übertragung der AGM-Revisionspostulate in [43] von deduktiv abgeschlossenen Formelmengen auf endliche aussagenlogische Formelmengen übertragen die Autoren in [46] die Kontraktionspostulate. Analog zu (R1) - (R6) für die Revision auf endlichen aussagenlogischen Formelmengen werden sieben Postulate (C1) - (C7) für die Kontraktion auf endlichen aussagenlogischen Formelmengen definiert. Dabei wird eine Korrespondenz zwischen einem Kontraktionsoperator auf Belief Sets $K \div \mu$ und auf aussagenlogischen Formeln $\psi - \mu$ hergestellt:

(C1) μ *implies* $\psi - \mu$
(C2) *if* ψ *does not imply* μ, *then* $\psi - \mu$ *implies* ψ
(C3) *if* $\psi - \mu$ *implies* μ, *then* $\varnothing$ *implies* μ
(C4) $(\psi - \mu) \wedge \mu$ *implies* ψ
(C5) *if* $\psi_1 \equiv \psi_2$ *and* $\mu_1 \equiv \mu_2$ *then* $\psi_1 - \mu_1 \equiv \psi_2 - \mu_2$
(C6) $\psi - (\mu \wedge \phi)$ *implies* $(\psi - \mu) \vee (\psi - \phi)$
(C7) *if* $\psi - (\mu \wedge \phi)$ *does not imply* μ, *then* $\psi - \mu$ *implies* $\psi - (\mu \wedge \phi)$

Proposition 4 (Überführung der AGM-Kontraktionspostulate [46]) *Ein Kontraktionsoperator $-$, erfüllt die Bedingungen (C1) - (C7), gdw. sein zugehöriger Kontraktionsoperator auf deduktiv abgeschlossenen Belief Sets $\div$ die acht AGM-Postulate (G1) - (G8) für die transitively relational partial meet contraction erfüllt.*

- (C1) entspricht dabei der Forderung, dass durch die Kontraktion kein neues Wissen in die aktualisierte Wissensbasis aufgenommen werden darf.
- (C2) fordert, dass wenn die neue Information nicht aus der Wissensbasis abgeleitet werden kann, durch die Kontraktion keine Änderung der Wissensbasis durchgeführt wird.

- (C3) fordert, dass die einzige Möglichkeit einer für eine fehlgeschlagene Kontraktion eine Kontraktion mit einer Tautologie ist.
- (C4) sagt aus, dass eine Kontraktion einer Formel und die anschließende logische Konjunktion der entstandenen Wissensbasis mit der Formel wieder zur ursprünglichen Wissensbasis führt.
- (C5) beschreibt die Irrelevanz der Syntax.
- (C6) fordert, dass sich aus dem Ergebnis einer Kontraktion mit einer Konjunktion aus zwei Formeln immer die Disjunktion der Kontraktionen mit den zwei Formeln ableiten lässt.
- (C7) beschreibt das Verhalten, dass wenn während der Kontraktion mit einer Konjunktion von zwei Formeln eine Formel nicht entfernt wurde, die Kontraktion mit der Formel die Kontraktion mit der Konjunktion impliziert.

Durch die Levi und Harper Identität konnte für die AGM-Kontraktion und Revision auf deduktiv abgeschlossenen Belief States der enge Zusammenhang formalisiert werden. Die Autoren in [46] übertragen Levi und Harper Identität auf die Operatoren auf endlichen aussagenlogischen Wissensbasen.

Definition 16 (Levi und Harper Identität auf Wissensbasen [46]) *Die Übetragung von Levi und Harper Identität auf Kontraktions- und Revisionsoperatoren auf endlichen aussagenlogischen Formelmengen lautet wie folgt:*

$$\begin{aligned} \textbf{(Levi Identität*)} \quad & \psi \circ_{(-)} \mu = (\psi - \neg\mu) \wedge \mu \\ \textbf{(Harper Identität*)} \quad & \psi -_{(\circ)} \mu = \psi \vee (\psi \circ \neg\mu) \end{aligned}$$

Es kann gezeigt werden, dass dadurch der Zusammenhang zwischen den Postulaten der Kontraktion (C1) - (C7) und denen der Revision (R1) - (R6) auf Wissensbasen hergestellt werden kann.

Proposition 5 (Levi und Harper Identität auf Wissensbasen [46]) *Ein durch die Levi Identität konstruierter Revisionsoperator $\circ_-$ erfüllt die Postulate (R1) - (R4), wenn der zu Grunde liegende Kontraktionsoperator $-$ (C1) - (C5) erfüllt.*
Ein durch die Levi Identität konstruierter Revisionsoperator $\circ_-$ erfüllt die Postulate (R1) - (R6), wenn der zu Grunde liegende Kontraktionsoperator $-$ (C1) - (C7) erfüllt.

Ein durch die Harper Identität konstruierter Kontraktionoperator $-_\circ$ erfüllt die Postulate (C1) - (C5), wenn der zu Grunde liegende Revisionsoperator $\circ$ (R1) - (R4) erfüllt.
Ein durch die Harper Identität konstruierter Kontraktionoperator $-_\circ$ erfüllt die Postulate (C1) - (C7), wenn der zu Grunde liegende Revisionsoperator $\circ$ (R1) - (R6) erfüllt.

Analog zum Repräsentationstheorem für die Revision, kann ein solches für die Erfüllung der Postulate (C1) - (C7) für die Kontraktion aufgestellt werden.

Proposition 6 (Repräsentationtheorem Kontraktion [46]) *Wenn eine totale Quasiordnung $\leq_\psi$ für Interpretationen existiert und die Zuordnung zu jeder Wissensbasis den Bedingungen für ein faithful assignment genügt, so dass $Mod(\psi - \mu) = Mod(\psi) \cup Min(Mod(\neg\mu), \leq_\psi)$ gilt, erfüllt der dadurch definierte Revisionsoperator die AGM-Postulate für die transitively relational partial meet contraction, d.h. die AGM-Revisionspostulate (G1) - (G8).*

$Min(Mod(\neg\mu), \leq_\psi)$ meint auch hier die kleinsten Interpretationen unter der angegebenen Ordnungsrelation von der angegebenen Menge von Interpretationen (hier $Mod(\neg\mu)$).

2.4 Iterated Belief Revision

Das originale AGM-Paper [44] ist eines der meist zitierten Veröffentlichungen im Bereich der Belief Revision. Viele Autoren haben sich seitdem mit dem Framework beschäftigt und entsprechend wurden auch einige Probleme mit den Postulaten durch weitere darauf aufbauende Arbeiten aufgedeckt. Zu nennen ist an der Stelle z.B. das häufig kritisierte Success Postulat für die Revision ($p \in K * p$), das fordert, dass neues Wissen immer Vorrang hat. In der Realität würde neues Wissen ggf. auch abgelehnt werden. Eine ausführliche Übersicht von Schwachstellen und Erweiterungen des AGM-Frameworks ist in [47] zu finden.
In dieser Arbeit soll allerdings ausschließlich auf das Problem der Iterated Belief Revision genauer eingegangen werden. Die AGM-Postulate sind nicht ausreichend, um kontraintuitive Resultate bei wiederholter Revision einer Wissensbasis zu verhindern. Beispiele von Operatoren, die zwar die AGM-Postulate erfüllen, aber zu kontraintuitiven Schlüssen bei mehrmaliger Revision einer Wissensbasis führen sind in [38] zu finden. Hauptkritik dabei ist die Tatsache, dass die AGM-Postulate für die Änderung von bedingtem Wissen (Hypothesen) kaum Einschränkungen bieten. Das lässt darauf zurückführen, dass die AGM-Postulate ausschließlich für einzelne Revisionsschritte formuliert wurden und bedingtes Wissen dadurch nicht betrachtet werden kann.
Der Lösungsvorschlag zu diesem Problem von Darwiche und Pearl in [38], welcher in den nächsten Kapiteln genauer vorgestellt wird, zielt auf die zusätzliche Einschränkung der Änderung, bzw. verstärkte Erhaltung von bedingtem Wissen ab. Dafür werden Revisionen nicht wie bisher auf Belief Sets durchgeführt, sondern auf sogenannten Epistemischen Zuständen. Darauf können Zwei-Schritt-Postulate als Erweiterung zu den auf Epistemische Zustände übertragenen AGM-Postulaten formuliert werden, um das Prinzip der „minimalen Änderung“ auch für bedingtes Wissen einzuführen.

2.4.1 Epistemische Zustände

Epistemische Zustände Ψ stellen die Erweiterung von Belief Sets dar. Sie enthalten zwar jeweils ein Belief Set, allerdings sind sie durch das Belief Set selbst noch nicht eindeutig definiert. Es existieren also mehrere Epistemische Zustände mit dem gleichen zu Grunde liegendem Belief Set. Die Notwendigkeit der Erweiterung sehen die Autoren deshalb, da für wiederholte Wissensrevision nicht nur das Verhalten für die Revision angegeben werden muss, sondern auch definiert werden muss, wie sich das Revisionsverhalten für weitere Revisionen ändern soll. Dies wird festgelegt durch die zusätzliche Beschränkung der Änderung von bedingtem Wissen in Epistemischen Zuständen. Der Umgang mit dem hypothetischen bedingten Wissen ist für die zukünftige Weltanschauung eines Agenten ebenso wichtig wie die Plausibilitätsordnung des vorhandenen Wissens. Deshalb sollte das bedingte Wissen ebenfalls entsprechend der Plausibilität geordnet werden.
Die Postulate (R1) - (R6) aus [43] werden auf die Epistemischen Zustände als (R*1) - (R*6) übertragen durch Ersetzen der Belief Sets ψ durch Epistemische Zustände Ψ. Dabei ist sobald Ψ in einer aussagenlogischen Formel verwendet wird, das zum Epistemischen Zustand gehörende Belief Set gemeint, d.h. vor allem $\Psi \equiv \Phi$ meint die Äquivalenz der Belief Sets von Ψ und Φ:

(R*1) $\Psi \circ \mu$ *implies* μ
(R*2) *if* $\Psi \wedge \mu$ *is satisfiable, then* $\Psi \circ \mu \equiv \Psi \wedge \mu$
(R*3) *if* μ *is satisfiable, then* $\Psi \circ \mu$ *is also satisfiable*
(R*4) *if* $\Psi_1 = \Psi_2$ *and* $\mu_1 \equiv \mu_2$ *then* $\Psi_1 \circ \mu_1 \equiv \Psi_2 \circ \mu_2$
(R*5) $(\Psi \circ \mu) \wedge \phi$ *implies* $\Psi \circ (\mu \wedge \phi)$
(R*6) *if* $(\Psi \circ \mu) \wedge \phi$ *is satisfiable, then* $\Psi \circ (\mu \wedge \phi)$ *implies* $(\Psi \circ \mu) \wedge \phi$

Neben dem Ersetzen von Belief Sets durch Epistemische Zustände muss für die Verwendung von Epistemischen Zuständen eines der Postulate abgeschwächt werden. (R4) wird ersetzt durch (R*4). (R4) fordert äquivalente Ergebnisse, bei Revision von zwei äquivalenten Belief Sets mit äquivalenter neuer Information. Das würde den Einsatz von Epistemischen Zuständen auf die zu Grunde liegenden Belief Sets reduzieren, da die Erweiterung der Belief Sets keine Rolle für die Revision spielen würde. Deshalb wird (R4) abgeschwächt, so dass äquivalente Ergebnisse nur noch gefordert werden, falls die beiden Epistemischen Zustände der Revision identisch sind. (R4) widerspricht außerdem dem in der Arbeit eingeführten Zwei-Schritt-Postulat (C*2) und ist laut den Autoren in [38] eine Ursache für kontraintuitive Revisionsergebnisse. Dafür geben Darwiche und Pearl auch konkrete Beispiele an.
Neben den Postulaten (R1) - (R6) zu (R*1) - (R*6) wird außerdem die bereits in Kapitel 2.3.4 eingeführte Definition des *faithful assignment* und das entsprechende Repräsentationstheorem aus [43] auf Epistemische Zustände, durch einfaches Ersetzen der

Belief States durch Epistemische Zustände, übertragen:

Definition 17 (faithful asisgnment für Epistemische Zustände) *Eine Funktion, die jeden Epistemischen Zustand Ψ in L einer totalen Quasiordnung $\leq_\Psi$ auf allen Welten Ω zuordnet, wird als faithful assignment bezeichnet, gdw. folgende drei Bedingungen zutreffen:*

$$\begin{array}{ll} \textbf{(1*)} & \omega_1, \omega_2 \models \Psi \;\Rightarrow\; \omega_1 =_\Psi \omega_2 \\ \textbf{(2*)} & \omega_1 \models \Psi \; and \; \omega_2 \not\models \Psi \;\Rightarrow\; \omega_1 <_\Psi \omega_2 \\ \textbf{(3*)} & \Psi = \Phi \;\Rightarrow\; \leq_\Psi = \leq_\Phi \end{array}$$

Proposition 7 (Repräsentationstheorem für Epistemische Zustände [38]) *Wenn eine totale Quasiordnung $\leq_\Psi$ für Interpretationen existiert und die Zuordnung zu jeder Wissensbasis den Bedingungen für ein faithful assignment genügt, so dass $Mod(\Psi \circ \mu) = Min(Mod(\mu), \leq_\Psi)$ gilt, erfüllt der dadurch definierte Revisionsoperator die Postulate (R*1) - (R*6).*

Damit ist das Repräsentationstheorem analog dem für die Postulate (R1) - (R6), mit der Ausnahme, dass für die Implikation von $\leq_\Psi = \leq_\Phi$ die Äquivalenz der Belief Sets von Ψ und Φ, d.h. $\Psi \equiv \Phi$ nicht ausreicht, sondern die Identität $\Psi = \Phi$ der Epistemischen Zustände gefordert wird.

2.4.2 Postulate von Darwiche und Pearl

Eine Möglichkeit den Umgang mit bedingtem Wissen bei iterierter Revision zu handhaben stellt das Zwei-Schritt-Postulat (CB) dar:

$$\text{(CB)} \; if \; \Psi \circ \mu \models \neg\alpha, \; then \; (\Psi \circ \mu) \circ \alpha \equiv \Psi \circ \alpha$$

Das Postulat lässt sich wieder gemäß dem eingeführten Repräsentationstheorem für epistemische Zustände in eine Bedingung für eine totale Quasiordnung auf Interpretationen überführen.

Proposition 8 (Repräsentationstheorem für (CB) [38]) *Gegeben sei ein Operator, der (R*1) - (R*6) erfüllt. Dieser erfüllt ebenfalls (CB), wenn er, bzw. das zugehörige faithful assignment, (CBR) erfüllt:*

$$\textit{(CBR)} \; if \; \omega_1, \omega_2 \models \neg(\Psi \circ \mu), \; then \; \omega_1 \leq_\Psi \omega_2 \;\Leftrightarrow\; \omega_1 \leq_{\Psi \circ \mu} \omega_2$$

Das Postulat fordert strikte Minimalität der Änderungen auch für bedingtes Wissen und ist somit der radikalste denkbare Ansatz für die Erhaltung von bedingtem Wissen. Das führt allerdings genauso wie eine zu geringe Einschränkung der Änderungen zu kontraintuitiven Schlüssen, da jetzt teilweise bereits erhaltenes Wissen im nächsten Revisionsschritt ohne das Vorliegen von widersprüchlicher Information verworfen wird, damit bedingtes Wissen erhalten werden kann.
Stattdessen wird das Postulat (CB) in [38] in sechs Zwei-Schritt-Postulate (C*1) - (C*6) auf Basis der verschiedenen Arten von bedingtem Wissen aufgeteilt:

(C*1) *if* $\alpha \models \mu$, *then* $(\Psi \circ \mu) \circ \alpha \equiv \Psi \circ \alpha$
(C*2) *if* $\alpha \models \neg\mu$, *then* $(\Psi \circ \mu) \circ \alpha \equiv \Psi \circ \alpha$
(C*3) *if* $\Psi \circ \alpha \models \mu$, *then* $(\Psi \circ \mu) \circ \alpha \models \mu$
(C*4) *if* $\Psi \circ \alpha \not\models \neg\mu$, *then* $(\Psi \circ \mu) \circ \alpha \not\models \neg\mu$

(C*5) *if* $\Psi \circ \mu \models \neg\alpha$ *and* $\Psi \circ \alpha \not\models \mu$, *then* $(\Psi \circ \mu) \circ \alpha \not\models \mu$
(C*6) *if* $\Psi \circ \mu \models \neg\alpha$ *and* $\Psi \circ \alpha \models \neg\mu$, *then* $(\Psi \circ \mu) \circ \alpha \models \neg\mu$

Der Vorschlag von Darwiche und Pearl besteht darin, nur die vier Postulate (C*1) - (C*4) für die iterierte Revision heranzuziehen. (C*5) und (C*6) hingegen nicht, da diese für das kontraintuitive Verhalten von Operatoren, die (CB) erfüllen, verantwortlich sind. Durch die Ergänzung der AGM-Postulate um (C*1) - (C*4) wird dem Prinzip der Änderungsminimierung auch für bedingtes Wissen gefolgt, allerdings nicht in der Radikalität, die die Ursache für das kontraintuitive Verhalten der Postulate (C*5) und (C*6), bzw. (CB) darstellt.
In [38] wird gezeigt, dass sich keines der Postulate (C*1) - (C*4) aus den AGM-Postulaten (G*1) - (G*8) ableiten lässt. Durch die Erweiterung sind nun Beziehungen zwischen $\leq_\Psi$ und $\leq_{\Psi\circ\mu}$ vorhanden, die aus den AGM-Postulaten nicht hervorgehen. Das wird deutlich durch die Übertragung von (C*1) - (C*4) auf die Repräsentation von Revisionen durch totale Ordnungen.

Proposition 9 (Repräsentationstheorem für (CR1) - (CR4) [38]) *Gegeben sei wieder ein Operator, der die Postulate (R*1) - (R*6) erfüllt. Dieser Operator erfüllt ebenfalls (C*1) - (C*4), gdw. er, bzw. das zugehörige faithful assignment, (CR1) - (CR4) erfüllt:*

(CR1) *if* $\omega_1 \models \mu$ *and* $\omega_2 \models \mu$, *then* $\omega_1 \leq_\Psi \omega_2 \Leftrightarrow \omega_1 \leq_{\Psi \circ \mu} \omega_2$
(CR2) *if* $\omega_1 \models \neg\mu$ *and* $\omega_2 \models \neg\mu$, *then* $\omega_1 \leq_\Psi \omega_2 \Leftrightarrow \omega_1 \leq_{\Psi \circ \mu} \omega_2$
(CR3) *if* $\omega_1 \models \mu$ *and* $\omega_2 \models \neg\mu$, *then* $\omega_1 <_\Psi \omega_2 \Rightarrow \omega_1 <_{\Psi \circ \mu} \omega_2$
(CR4) *if* $\omega_1 \models \mu$ *and* $\omega_2 \models \neg\mu$, *then* $\omega_1 \leq_\Psi \omega_2 \Rightarrow \omega_1 \leq_{\Psi \circ \mu} \omega_2$

Ein Beispiel für einen Operator, der (R1) - (R6) sowie (CR1) - (CR4) erfüllt, wird von Darwiche und Pearl ebenfalls angegeben. Dabei handelt es sich um eine abgewandelte Version des Revisionsoperators von Spohn [48] [49], welcher auf einem Ranking der Interpretationen nach Plausibilität basiert.

3 Änderungsräume

Zur Beschreibung von Wissensänderungsoperatoren wird als Struktur der sogenannte Änderungsraum definiert. Ähnlich wie ein Graph besteht dieser aus einer Menge von Knoten und Kanten. Dabei repräsentieren die Knoten die verschiedenen Zustände, welche jeweils mit einer totalen Quasiordnung verknüpft sind. Die Kanten entsprechen lokalen Wissensänderungen und besitzen somit neben dem Anfangs- und Endknoten die neue Information, mit der die Wissensänderung durchgeführt wird. Angelehnt ist die Darstellung an eine Veröffentlichung von Aravanis, Peppas und Williams aus 2019 [3].

Definition 18 (Änderungsraum) *Sei Ω eine endliche Menge von Welten. Ein* Änderungsraum *über Ω ist ein Tupel $\mathbb{C} = (S, C, \tau, \ell)$, so dass*

- *(S, C) ein (gerichteter) Graph mit Knotenmenge S und Kantenmenge C ist, wobei die Elemente von S als Zustände bezeichnet werden und die Elemente von C als Änderungen,*
- *τ ordnet jedem Zustand $s \in S$ eine totale Quasiordnung $\tau(s) \subseteq \Omega \times \Omega$ zu, und*
- *ℓ ordnet jeder Änderung $(s_1, s_2) \in C$ eine Teilmenge $\ell(s_1, s_2) \subseteq \Omega$ zu.*

Man beachte, dass im Allgemeinen in einem Änderungsraum der Zustand eines Agenten $s \in S$ und die Ordnung $\tau(s)$ nicht gleich sind. Diese Erweiterung der iterierten Revision schlagen die Autoren in [3] vor, denn sie kommen zu dem Schluss, dass die eindeutige Beschreibung der Zustände der Agenten ausschließlich über die Ordnungen nicht ausreichend ist. Es können nicht alle möglichen Revisionsoperatoren dargestellt werden, die den Postulaten von Darwiche und Pearl, siehe Kapitel 2.4.2, genügen. Abbildung 4 gibt Beispiele für Änderungsräume:

(a) Der Änderungsraum $\mathbb{C}_1$ (b) Der Änderungsraum $\mathbb{C}_2$

Abbildung 4: Beispiele für Änderungsräume. Angelehnt an [3].

- Ein Änderungsraum heißt *endlich*, falls S eine endliche Menge ist.
- Ein Änderungsraum wird als *deterministisch* bezeichnet, falls es für jedes $s_1 \in S$ und jede Menge $\alpha \subseteq \Omega$ höchstens ein $s_2 \in S$ gibt mit $\ell(s_1, s_2) = \alpha$.
- Ein Änderungsraum $\mathbb{C}^* = (S^*, C^*, \tau^*, \ell^*)$ *enthält einen Änderungsraum* $\mathbb{C} = (S, C, \tau, \ell)$ *komplett*, kurz $\mathbb{C} \sqsubseteq \mathbb{C}^*$, wenn $S \subseteq S^*$ und $C \subseteq C^*$, und $\tau(s_1) = \tau^*(s_2)$ und $\ell^*(s_1, s_2) = \ell(s_1, s_2)$ für alle $s_1, s_2 \in S$.
  Beispielsweise enthält $\mathbb{C}_2$ aus Abbildung 4 den Änderungsraum $\mathbb{C}_1$ aus Abbildung 4 komplett.
- Des Weiteren wird ein Änderungsraum (S, C, τ, ℓ) bezüglich Ω als *vollständig* bezeichnet, falls folgende zwei Bedingungen zutreffen:
  - Für jede totale Quasiordnung $\preceq \subseteq \Omega \times \Omega$ gibt es einen Zustand $s \in S$, so dass $\tau(s) = \preceq$ gilt.
  - Zu jedem Zustand $s \in S$ und jeder Teilmenge $\Omega' \subseteq \Omega$ gibt es einen Zustand s^*, so dass $(s, s^*) \in C$ und $\ell(s, s^*) = \Omega'$ gilt.

  Beide gezeigten Änderungsräume $\mathbb{C}_1$ und $\mathbb{C}_2$ in Abbildung 4 sind also nicht vollständig. Sie sind Beispiele für partiell definierte Änderungsräume als Repräsentation für partiell spezifizierte Revisionsoperatoren.

4 FO-BC Signatur

Für die Formulierung von Bedingungen an Wissensänderungen wird eine Spezifikationssprache benötigt. Hierfür wird die first order logic for belief changes (FO-BC) eingeführt, eine eingeschränkte Version der Prädikatenlogik erster Stufe.

Definition 19 (FO-BC Signatur) *Eine FO-BC Signatur* $\sum = (Func, Pred)$ *besteht aus drei Prädikatensymbolen und einem Funktionssymbol. Das einzige Funktionssymbol c in der Menge der Funktionssymbole Func der Signatur ist null-stellig, es handelt sich also um eine Konstante.*
Die Menge der Prädikatensymbole Pred enthält die folgenden drei Symbole: $Pred = \{\preceq_1 /2,\ \preceq_2 /2, Mod_\alpha/1\}$.

Auf Basis dieser Signatur lassen sich mit Hilfe der Operatoren und Quantoren der Prädikatenlogik erster Stufe Formeln aufstellen, die Bedingungen an einzelne Wissensänderungen darstellen. Das zweistellige Prädikatensymbol $\preceq_1$ wird interpretiert zu allen zweiwertigen Elementpaaren, für die das erste Argument bezüglich der totalen Quasiordnung über alle möglichen Welten der Ausgangswissensbasis kleiner bzw. gleichrangig dem zweiten Argument ist. Das ebenfalls zweistellige Prädikatensymbol $\preceq_2$ wird analog zu den Paaren an Elementen des Universums interpretiert, für die das erste Argument bezüglich der totalen Quasiordnung über den möglichen Welten nach der Wissensänderung kleiner oder gleichranging dem zweiten Argument ist. Das dritte Prädikatensymbol Mod_α ist einstellig und wird interpretiert als die Menge aller Welten, die der an der Wissensänderung beteiligten neuen Information zugehören.

Mit Hilfe der FO-BC Signatur lassen sich beispielsweise die im Kapitel 2.4.2 aufgeführten Beispiele für Postulate formulieren. Als Beispiel dient hier das Postulat CR1 von Darwiche und Pearl [38]:

$$\textit{if } \omega_1, \omega_2 \in Mod(\alpha), \textit{ then } \omega_1 \preceq_1 \omega_2 \textit{ if and only if } \omega_1 \preceq_2 \omega_2$$

Überführt in eine FO-BC Formel in Infix Schreibweise lautet das Postulat wie folgt:

$$\phi_{CR1} = \forall\omega_1\forall\omega_2((\omega_1 \in Mod(\alpha) \wedge \omega_2 \in Mod(\alpha)) \rightarrow (\omega_1 \preceq_1 \omega_2 \leftrightarrow \omega_1 \preceq_2 \omega_2))$$

Um die eingeführten Änderungsräume, die Repräsentationsstruktur von Operatoren, mit den FO-BC Bedingungen zu verbinden, wird die Erfüllungsrelation $\models$ zwischen einer FO-BC Formel ϕ und einem Änderungsraum $\mathbb{C}$ wie folgt definiert:

Definition 20 (Erfüllungsrelation auf Änderungsräumen) *Sei $\mathbb{C} = (S, C, \tau, \ell)$ ein Änderungsraum und φ eine Formel über einer FO-BC-Signatur.*
Für jedes $(s_1, s_2) \in C$ wird die Interpretation

$$\mathcal{A}^{\mathbb{C}}_{s_1,s_2} = (\Omega, \{c^{\mathcal{A}_{s_1,s_2}}\}, \{\tau(s_1), \tau(s_2), \ell(s_1, s_2)\})$$

definiert, wobei $c^{\mathcal{A}_{s_1,s_2}}$ ein beliebiges Objekt aus Ω ist.
Es sei $\mathbb{C} \models \varphi$, falls $\mathcal{A}^{\mathbb{C}}_{s_1,s_2} \models \varphi$ für alle $(s_1, s_2) \in C$.

In Worten bedeutet das, dass für jede mögliche lokale Wissensänderung innerhalb des gegebenen Änderungsraums die FO-BC Bedingung erfüllt sein muss, damit die Formel vom Änderungsraum erfüllt wird. Das macht sich der Algorithmus zu nutze, indem er bei der Vervollständigung mit Hilfe der Brute-Force-Methodik jede fehlende Kante auf diese Bedingung hin überprüft, bis eine Kante gefunden wurde, bei der diese erfüllt ist. Die gefundene Kante wird dann dem Änderungsraum hinzugefügt.

5 Vervollständigung von Wissensoperatoren

Im Rahmen der Arbeit wird ein Programm erstellt, welches partiell spezifizierte Wissensoperatoren, wie in Kapitel 3 eingeführt, vervollständigt. Für die Vervollständigung können mit Hilfe der in Kapitel 4 definierten FO-BC Signatur Bedingungen definiert werden.
Bei dem zu Grunde liegenden Problem handelt es sich um ein Ω-Vervollständigungsproblem. Der Algorithmus, welcher für die Lösung des Problems implementiert wird, verwendet eine Brute-Force-Methodik:

Algorithmus 1 : Computing the completion for a change space

1: **Funktion** `completion` **mit**
   Input : Deterministischer endlicher Änderungsraum $\mathbb{C}$ über Ω und eine Formel φ über einer FO-BC-Signatur, so dass $\mathbb{C} \models \varphi$ gilt.
   Output : Ein vollständiger deterministischer endlicher Änderungsraum $\mathbb{C}^*$ über Ω, der $\mathbb{C}$ komplett enthält und für den $\mathbb{C}^* \models \varphi$ gilt. Fehlermeldung, falls ein solcher Änderungsraum nicht existiert.
2: $S^* \leftarrow S$ **and** $C^* \leftarrow C$ **and** $\tau^* \leftarrow \tau$ **and** $\ell^* \leftarrow \ell$
3: **foreach** *total preorder* $\leq$ *over* Ω **do**
4: **if** *there is no* $s \in S$ *with* $\preceq_s = \leq$ **then**
5: $S^* \leftarrow S^* \cup \{s^*\}$ `// where` s^* `is a fresh state`
6: $\tau^* \leftarrow \tau^* \cup \{s^* \mapsto \leq\}$
7: **foreach** $s \in S^*$ **and** $\alpha \subseteq \Omega$ **do**
8: **if** *there is no* $s^* \in S^*$ *with* $(s, s^*) \in C^*$ *and* $\ell^*(s, s^*) = \alpha$ **then**
9: **if** *there is* $s^* \in S^*$ *with* $\mathcal{A}_{s,s^*}^{(S^*,C^*,\tau^*,\ell^*)} \models \varphi$ **then**
10: $C^* \leftarrow C^* \cup \{(s, s^*)\}$
11: $\ell^* \leftarrow \ell^* \cup \{(s, s^*) \mapsto \alpha\}$
12: **else**
13: **return** *no*
14: **return** (S, C, τ, ℓ) `// return change space` $\mathbb{C}^*$

Der Algorithmus fügt zunächst fehlende Zustände in der ersten Schleife (Zeile 3 bis 6) ein und dann fehlende Änderungen in der zweiten Schleife (Zeile 7 bis 13). Folgende Eigenschaften, die aus der Beschreibung des Algorithmus hervorgehen, sind besonders hervorzuheben:

- **Determinismus von Ein- und Ausgabeänderungsräumen**: Der Algorithmus ist für deterministische Änderungsräume definiert. Nach der Definition aus Kapitel 3 existiert damit für jede Kombination aus Ausgangszustand und neuer Information nur eine Kante im Graph. Der geforderte Determinismus des Ein-

gabeänderungsraums wird nicht mehr explizit geprüft. Falls der Determinismus nicht gegeben ist, ist die Ausgabe des Algorithmus ebenfalls nicht-deterministisch.

- **Erhalt des Eingabeänderungsraums**: Der eingegebene (in der Regel partiell spezifizierte) Änderungsraum wird durch den Algorithmus lediglich erweitert. Nach der Beschreibung des Algorithmus enthält der Änderungsraum der Ausgabe den der Eingabe komplett. Bereits vorhandene Kanten des Eingabegraphen werden nicht auf die eingegebenen FO-BC Bedingungen geprüft. Es wird $\mathbb{C} \models \varphi$ nach der Definition für die Eingabe vorausgesetzt.

5.1 Software Architektur

Neben der Umsetzung des Kernalgorithmus sind für die Erstellung des kompletten Programmes weitere Komponenten erforderlich. Das Programm wird in der Programmiersprache C++ umgesetzt. Da C++ eine kompilierte Programmiersprache ist und keinen Interpreter benötigt, kann ein effizienteres Programm erstellt werden, das nur einmal initial kompiliert werden muss. Bei der Erstellung wurde darauf geachtet, dass die Software sowohl unter Windows, als auch Linux auführbar ist.
Wie in der C++ Programmierung üblich für größere Programme wird die Software in verschiedene Dateien aufgeteilt, um den Code übersichtlicher zu machen. Davon profitiert auch eine spätere Weiterentwicklung bzw. Wiederverwendung des Codes bzw. Teilen des Codes in anderen Arbeiten. Die Aufteilung der Software in die verschiedenen Bestandteile mit den jeweilig wichtigsten Funktionen der Module ist in Abbildung 5 zu sehen.

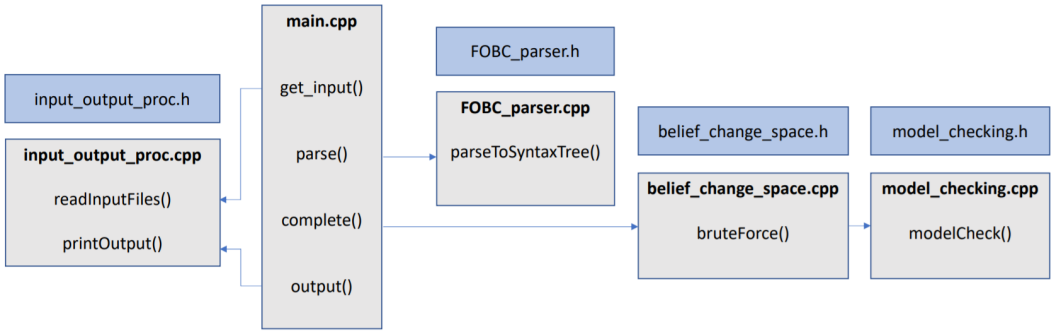

Abbildung 5: Softwarearchitektur des Programms (C++)

Die Header-Dateien (.h) enthalten die Funktionsdefinitionen und ermöglichen durch Einbindung in die anderen C++-Dateien (.cpp) den Aufruf der Funktionen. Die eigentliche Implementierung der Funktionen ist in den jeweiligen C++-Dateien zu finden.
Der Ablauf des Programms ist wie folgt:

- **Einlesen der Eingabedateien**: Zum Einlesen der Eingaben des Algorithmus wird auf Funktionen aus dem Modul *input_output_proc.cpp* zugegriffen. Diese

liefern eine Repräsentation des eingegebenen Änderungsraums, sowie der eingegebenen FO-BC Formeln.

- **Erzeugen von Syntaxbäumen**: Für jede eingelesene FO-BC Formel wird anschließend mit Hilfe des Moduls *FOBC_parser.cpp* ein *SyntaxTree* Objekt erstellt und die Formeln durch Aufruf der Methode *parse()* in Syntaxbäume umgewandelt.

- **Vervollständigung**: Nach Erstellung eines *Change Operator* Objekts wird mit Hilfe der Methode *complete()* mit den erzeugten Syntaxbäumen als Argument der Kernalgorithmus ausgeführt. Die Brute-Force-Methodik in beiden Schleifen, d.h. erst Einfügen der fehlenden Knoten und anschließend Einfügen der fehlenden Kanten des Graphen ist im Modul *belief_change_space.cpp* umgesetzt. Es greift für die Überprüfung der Erfüllungsrelation zwischen FO-BC Formeln und Interpretationen auf das Modul *model_checking.cpp* zurück.

- **Ausgabe des Ergebnisses**: Der vervollständigte Änderungsraum wird zuletzt durch das Input-/Output-Modul (I/O) *input_output_proc.cpp* ausgegeben. Falls die Vervollständigung des Algorithmus nicht erfolgreich war, erzeugt das Programm entsprechend eine Fehlermeldung.

Im Folgenden wird die Umsetzung der einzelnen Software-Module detaillierter ausgeführt. Zusätzlich existiert eine Bedienungsanleitung für die Benutzung der Software in Form einer *README*-Textdatei, welche dem Code beigefügt ist. Um die Leserlichkeit zu verbessern, sind die Code-Ausschnitte nicht direkt im Text eingebunden. Wenn benötigt, werden ausgewählte Ausschnitte im Kapitel 7 referenziert. Der komplette Source-Code ist im Anhang der Arbeit zu finden.
Die Übergabe von Daten zwischen Funktionen ist im gesamten Programm ausschließlich über Referenzen bzw. Pointer gelöst, wodurch der Code sehr effizient ausgeführt werden kann.

5.1.1 Input-/Output-Processing

Die Programmeingabe erfolgt mit Hilfe von drei Dateien. In zwei Dateien wird der zu vervollständigende Änderungsraum definiert. In der dritten Datei werden die FO-BC Bedingungen für die Vervollständigung formuliert.

Für die Eingabe des Änderungsraums muss nicht nur ein Graph durch Knoten und Kanten spezifiziert werden, sondern auch die jeweilige totale Quasiordnung für jeden Knoten des Graphen definiert und die Kanten der jeweiligen Teilmenge von allen betrachteten Welten zugeordnet werden. Als praktische Lösung zu dieser Problemstellung

ist die Eingabe auf zwei Tabellen (*.csv* Dateien) aufgeteilt. Angelehnt sind die Tabellen an die programminterne Repräsentation des Änderungsraums (siehe 5.1.3) und ermöglichen so eine möglichst einfache Verarbeitung der Eingabe.

In einer ersten Tabelle (siehe Abbildung 6) werden die Knoten des Graphen mit der zugewiesenen totalen Quasiordnung spezifiziert. Die gewählte Darstellung basiert auf der Vergabe von Plausibilitätsrängen in Form von natürlichen Zahlen einschließlich der Null. Dabei wird jeder Welt für jeden Zustand eine Zahl zugewiesen. Im Programm wird für die Ränge beim Einlesen eine Standardisierung durchgeführt (siehe Listing 12), d.h. selbst wenn größere Zahlen, als die Anzahl von Welten, eingegeben werden, kann eine totale Quasiordnung abgeleitet werden. Am konkreten Beispiel aus 6 ist im Zustand *s1* bezüglich der so definierten totalen Quasiordnung über drei Welten $w1 < w2$, $w1 < w3$ und $w2 < w3$.

	A	B	C	D	E	F
1	*Predefined States s with corresponding total preorders t(s) over worlds w					
2	*Insert plausibility index as natural numbers (0 is most probable)					
3						
4	States/Worlds	w1	w2	w3		
5	s0	0	0	0		
6	s1	0	1	2		
7						
8						
9						

Abbildung 6: Eingabedatei 1 (Vordefinierte Zustände und ihre totalen Quasiordnungen)

Die Namen für die Welten, als auch für die Zustände, können frei gewählt werden. Es können dabei grundsätzlich beliebig viele Knoten mit Quasiordnungen spezifiziert werden. Wichtig für das korrekte Einlesen ist nur, dass die durch die Beschriftungen der Zustände und Welten (im Beispiel 6 {*s0, s1*} und {*w0, w1, ...*}) aufgespannte Matrix mit den Dimensionen $n \times m$, wobei n der Anzahl an Zuständen und m der Anzahl an betrachteten Welten entspricht, vollständig mit natürlichen Zahlen befüllt ist. Aus den Längen der Beschriftungsvektoren in dieser Datei leitet das Programm n und m für die spätere Nutzung ab. Die Bezeichnungen müssen manuell je nach Anzahl an betrachteten Welten und vordefinierten Zuständen angepasst werden, um die benötigte Matrix aufzuspannen. Im Code sind zur Absicherung einige Abfragen für mögliche Fehleingaben hinterlegt, die zu einer Fehlermeldung während der Ausführung des Programms führen. Zeilen, die mit einem ASCII-Stern beginnen, werden beim Einlesen als Kommentar erkannt und genauso wie leere Zeilen ignoriert.

In einer zweiten Datei werden die vordefinierten Kanten des Eingabegraphen definiert (siehe Abbildung 7). Hier müssen die Anzahl und Bezeichnung der Welten aus der ers-

ten Eingabetabelle für die Zustände übernommen werden. Anschließend können über eine beliebige Anzahl von Zeilen verschiedene Kanten eingefügt werden. Eine Kante wird definiert über ihren Ausgangszustand $s1$, ihren Zielzustand $s2$ und die Teilmenge von allen betrachteten Welten $\ell(s_1, s_2) \subseteq \Omega$. Diese Teilmenge wird definiert über das Befüllen der Zeile in der aufgespannten Matrix mit *true*, für die Welten, die Teil von ℓ sind, und *false*, für die Welten, die nicht Teil von ℓ sind. Auch diese Matrix muss vollständig befüllt werden. Für Fehleingaben sind ebenfalls Absicherungen im Programm vorgesehen, welche zur Ausgabe einer Fehlermeldung führen.

	A	B	C	D	E	F	G	H
1	*Predefined Edges with corresponding worlds l(s1, s2)							
2	*Insert <true> for worlds, that are part of alpha for the corresponding edge, <false> for worlds, which are not							
3	*Make sure to at least have one <true> for each row, since edges without corresponding worlds are not accepted							
4	*make sure, that the states and worlds match the ones in the first input file							
5								
6	origin	destination/alpha	w1	w2	w3			
7	s0	s0	true	true	true			
8	s1	s0	false	false	true			
9								
10								

Abbildung 7: Eingabedatei 2 (Vordefinierte Kanten und ihre Modelle)

Es muss darauf geachtet werden, dass für die Angabe des Ausgangs- und Zielzustandes nur Bezeichnungen verwendet werden, die in der ersten Eingabedatei spezifiziert sind. Entsprechend der eingeführten Bedingung, dass Kanten mit leeren Modellen, d.h. $\ell(s_1, s_2) = \emptyset$, nicht existieren bzw. implizit wieder auf denselben Zustand führen, muss auch bei der Eingabe immer mindestens für eine der betrachteten Welten pro Zeile der eingetragene Wert *true* betragen. Im Beispiel 7 ist in der ersten Zeile eine Kante definiert, die vom Ausgangszustand *s1* zu *s0* geht mit $\ell(s0, s0) = w3$. Auch in dieser Datei werden leere Zeilen und Zeilen, die mit einem ASCII-Stern beginnen, beim Einlesen ignoriert.

Die dritte Eingabedatei kann mit den FO-BC Bedingungen für die Vervollständigung der Kanten des Graphen gefüllt werden. Es handelt sich um eine einfache Text-Datei (*.txt*), siehe Abbildung 1. Die Datei enthält Hinweise zur korrekten Eingabe von Formeln als Kommentare. Diese sind hier wieder dadurch gekennzeichnet, dass die Zeile mit einem Stern beginnt (*) und werden beim Einlesen der Datei ignoriert. Dadurch können in der Datei Formeln schnell und einfach auskommentiert werden.

Listing 1: Eingabedatei 3 (FO-BC Formeln)

```
1 * Define FO-BC conditions for completing the change operator
2 *
3 * FO-BC Signature consists of:
4 *       - constant c
5 *       - predicate Mod/1, written in prefix notation as Mod(x)
```

```
6 *                -> True, if argument x is part of Mod(a)
7 *        - predicate TPO1/2, written in prefix notation as TPO1(x,y)
8 *                -> True if argument x is smaller or equal y in total
    preorder of origin state of belief change
9 *        - predicate TPO2/2, written in prefix notation as TPO2(x,y)
10 *               -> True if argument x is smaller or equal y in total
    preorder of destination state of belief change
11 *
12 * Use following logical symbols:
13 *        &, |, ~, =>, <=>, A(x), E(x)
14 *
15 * Variables are defined as characters in range [u, v,..., z]
16 * If more variables are needed two numbers can be added (e.g. u1, x15,
    w79)
17 *
18 * Brackets can be used. Default binding definition:
19 *        (strongest) ~, &, |, =>, <=>, A, E (weakest)
20 *
21 * The algorithm can get rid of spaces in the sentences.
22 * You can enter multiple sentences.
23 *
24 *
25 *-----------------------------------------------------------------
26 *        Insert conditions below (one sentence per line)
27 *-----------------------------------------------------------------
28
29 *CR1
30 (Mod(x) & Mod(y)) => (TPO1(x,y) <=> TPO2(x,y))
31
32 *CR2
33 (~Mod(x) & ~Mod(y)) => (TPO1(x,y) <=> TPO2(x,y))
34 .....
```

Für die Eingabe der Bedingungen für die Wissensänderungen in das Programm, welche auf Basis der FO-BC Signatur in Form von Formeln beschrieben sind, wird eine Präfixschreibweise gewählt. Das vereinfacht das korrekte Einlesen durch den implementierten FO-BC Parser. Da außerdem keine Sonderzeichen zum Einsatz kommen sollen, um eine Lauffähigkeit des Programms ohne Modifikation auf möglichst vielen Plattformen zu ermöglichen, werden für die Eingabe der FO-BC Formeln die Prädikatensymbole durch reine alphanumerische Zeichenketten ersetzt.
Das Symbol $\preceq_1$ wird somit in Präfix-Schreibweise als $TPO1(x, y)$ notiert und das Symbol $\preceq_2$ als $TPO2(x, y)$. TPO ist hierbei die Abkürzung für die totale Quasiordnung (Total Pre-Order). Das dritte Symbol Mod_α wird zu $Mod(x)$ vereinfacht. Das Funktionssymbol c ist bereits ein reines Zeichen und muss deshalb für die vereinfachte Eingabe nicht ersetzt werden.

Neben den Prädikaten- und Funktionssymbolen muss die vereinfachte Eingabe auch für die Operatoren, sowie Quantoren der FOL gelten. Da sowohl das Zeichen für den Allquantor als auch Existenzquantor ein Sonderzeichen ist, welches nicht in allen Standard Zeichensätzen enthalten ist, wird hier auf die Präfix-Notation A(x) für den Allquantor bzw. E(y) für den Existenzquantor ausgewichen. Das Gleiche gilt für die bisher verwendete Schreibweise für die logischen Operatoren $\{\wedge, \vee, \rightarrow, \leftrightarrow\}$. Diese werden durch verbreitetere Zeichen gemäß Tabelle 4 ersetzt. Für Variablen wird ein gültiger Raum von alphanumerischen Kombinationen aus einem Buchstaben im Bereich $\{u, v, ..., z\}$ mit maximal zwei angehängten Zahlen definiert.

Symbol	Vereinfachte Eingabe für Parser	Bedeutung
$x \preceq_1 y$	$TPO1(x, y)$	Totale Quasiordnung Ausgangs-Wissensbasis
$x \preceq_2 y$	$TPO2(x, y)$	Totale Quasiordnung aktualisierte Wissensbasis
$x \in Mod_\alpha$	$Mod(x)$	Enthalten in neuer Information der Wissensänderung
$\neg$	$\sim$	Negation
$\wedge$	&	Konjunktion
$\vee$	|	Disjunktion
$\rightarrow$	=>	Materielle Implikation
$\leftrightarrow$	<=>	Äquivalenz
$\forall x$	$A(x)$	Allquantor
$\exists x$	$E(x)$	Existenzquantor
c	c	Konstantensymbol
$\{u, v, ..., z\}$	$\{u, v, ..., z)$, $(u1, u2..., u99)$, $(v1, v2, ..., v99)$, ..., $(z1, z2, ..., z99)\}$	Variablen

Tabelle 4: Vereinfachte Schreibweise für die Programmeingabe

Als Beispiel dient hier das Postulat CR1 in Infix-Schreibweise aus Kapitel 4:

$$\phi_{CR1} = \forall\omega_1\forall\omega_2((\omega_1 \in Mod(\alpha) \wedge \omega_2 \in Mod(\alpha)) \rightarrow (\omega_1 \preceq_1 \omega_2 \leftrightarrow \omega_1 \preceq_2 \omega_2))$$

In der vereinfachten Schreibweise für die Verarbeitung durch den FO-BC Parser lautet das Postulat wie folgt (siehe auch Listing 1):

$$A(w1)A(w2)(Mod(w1)\&Mod(w2)) => (TPO1(w1, w2) <=> TPO2(w1, w2))$$

Die Ausgabe des vervollständigten Graphen erfolgt im *.dot*-Format. Bei DOT handelt

es sich um eine Beschreibungssprache zur Visualisierung von Graphen. Die Ausgabe erfolgt nur, wenn ein vollständiger Änderungsraum berechnet werden konnte. In diesem Falle erstellt das Programm eine *.dot*-Datei, falls sie noch nicht vorhanden ist. In dieser sind die vordefinierten Teile des Änderungsraums blau gekennzeichnet und die vom Algorithmus eingefügten Teile besitzen eine graue Farbe. Für die vordefinierten Welten und Zustände werden die Bezeichnungen aus den Eingabedateien verwendet. Für neu hinzugefügte Zustände, d.h. Knoten im Graphen, werden die Namen $\{s0*, s1*, \dots\}$ verwendet.

Listing 2: Ausgabedatei (.dot Format)

```
digraph CompletedOperator{

rankdir=LR; concentrate=false;

subgraph Predef {
node[color=blue];
edge[color=blue];
"s0" -> "s0" [label=<<font color="blue">{w1, w2, w3}</font>>]
"s1" -> "s0" [label=<<font color="blue">{w3}</font>>]
node[color=blue shape=box style=dashed];
edge[color=blue style=dashed arrowhead=none];
"{w1, w2, w3}" -> "s0"
"{w1} < {w2} < {w3}" -> "s1"
}

node[style=filled];
edge[color=grey];
"s0" -> "s8*" [label="{w1}"]
"s0" -> "s4*" [label="{w2}"]
"s0" -> "s10*" [label="{w1, w2}"]
.....

}
```

Für die Visualisierung der Ausgabedatei muss die Datei von Renderern des *GraphWiz*-Pakets interpretiert werden. Hierzu finden sich einige Online-Implementierungen im World-Wide-Web. Ein Beispiel ist in der *README*-Datei aufgeführt. Listing 2 zeigt einen Ausschnitt aus einem berechneten vollständigen Änderungsraum für die Eingaben aus den Abbildungen 6, 7 und Listing 1. Abbildung 8 eine Visualisierung dazu.

Für das Handling der Ein- und Ausgaben wird ein C++-*Namespace* definiert (siehe Code Listing 4). In der *Main*-Funktion des Programms erfolgt dann nur ein einfacher Aufruf der zur Verfügung gestellten Funktionen im *fileproc::-Namespace*.

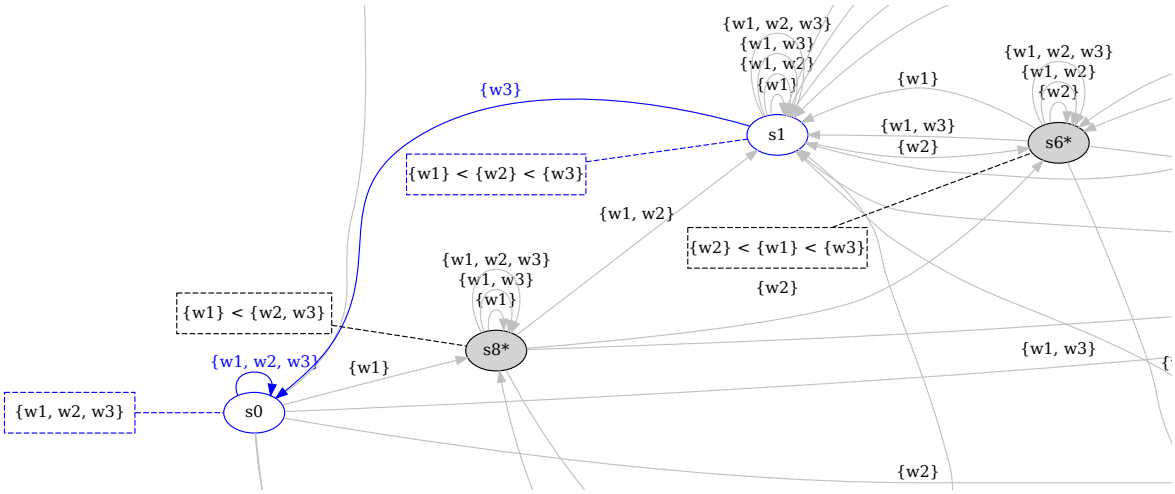

Abbildung 8: Interpretierte Ausgabedatei (.dot Format)

5.1.2 FO-BC Parser

Der implementierte Parser ist auf dem Code eines Inferenzprogrammes für die Prädikatenlogik erster Stufe, der „First-Order-Logic-Inference-Engine“ von V. Kadam[1], aufgebaut. Relevant für diese Arbeit ist allerdings nur der Teil des Codes, in dem der Parser umgesetzt ist. Dieser kann zwar bereits prädikatenlogische Formeln mit zweistelligen Prädikaten, Funktionssymbolen, Junktoren und Variablen parsen, allerdings nur ohne die Verwendung von Quantoren in den Formeln. Um diese Funktion wird der Parser für den Einsatz in der Arbeit erweitert und anschließend für das Logik Fragment FO-BC angepasst.
Grundlage dieses Moduls bildet die eingeführte Klasse *SyntaxTree*. Die Struktur der Klasse mit ihren Methoden ist in der Klassendefinition in Code Listing 5 zu sehen. Das Parsen der FO-BC Formeln erfolgt als Methode der erstellten Klasse *SyntaxTree* in zwei Schritten, siehe Code Listing 6.

- **Tokenisierung**: In diesem Schritt wird jede der Formeln, die jeweils als Zeichenketten eingelesen werden, unterteilt in die relevanten syntaktischen Blöcke, die sogenannten Token. Es wird ein Token pro Prädikat, Junktor, Klammer und Quantor erzeugt.

- **Generierung der Syntaxbäume**: Die Generierung der Syntaxbäume muss anhand der festgelegten Bindungsprioritäten erfolgen. Die Operation mit der geringsten Bindungspriorität bildet die Wurzel eines Syntaxbaums. Weiter unten im verzweigten Baum finden sich die Operationen mit höheren Bindungsprioritäten. Die Bindungsprioritäten außerhalb von explizit gesetzten Klammern sind für die Implementierung wie in Tabelle 5 gezeigt festgelegt.

[1] https://github.com/vritvij/First-Order-Logic-Inference-Engine (Commit 13b0618 vom 03.03.2017)

$\neg$	stärkste Bindungspriorität
$\wedge$	...
$\vee$	...
$\rightarrow$	...
$\leftrightarrow$	...
$\forall$	...
$\exists$	schwächste Bindungspriorität

Tabelle 5: Bindungsprioritäten Junktoren und Quantoren

Im Code ist dies umgesetzt durch Zuweisung einer ganzen Zahl zu jedem Operator über eine Funktion, die die Bindungspriorität repräsentiert (siehe Code Listing 7).
Um einen Syntaxbaum zu beschreiben wird eine Struktur für Knoten namens *node* angelegt, welcher die einzelnen Verzweigungspunkte darstellt. Jeder Knoten enthält Informationen über den jeweiligen Typ (Quantor, Junktor, Prädikat) und zwei Zeiger auf die nächsten zwei Elemente unterhalb der Verzweigung im Teilbaum. Die Blätter des Baums werden durch die Prädikat-Knoten dargestellt, welche zwei Null-Zeiger hinterlegt haben und zusätzliche Informationen über die Funktionssymbole besitzen. Die Quantoren und die logische Negation besitzen als rechten Zeiger einen Null-Pointer. Alle anderen Junktoren besitzen zwei Argumente und deshalb sind auch beide Zeiger befüllt. Wenn der Zeiger auf einen Wurzelknoten (*root*) eines Syntaxbaums übergeben wird, kann damit auf den gesamten Syntaxbaum zugegriffen werden. Die Generierung des Syntaxbaums erfolgt nun Stack-basiert. Dabei werden die Operator-Token, d.h. Junktoren, Quantoren und Klammern, so lange auf einen Stack gelegt, bis ein Token mit geringerer Bindungspriorität eintrifft. Dann werden erst die Operationen mit höherer Bindungspriorität ausgeführt und ein Wurzelknoten der Operation als neuer Token auf den zweiten Ergebnis Stack gelegt. Der Baum wird also von unten zur Wurzel hin aufgebaut.
Während der Erstellung des Syntaxbaums werden die einzelnen Knotenpunkte auf Plausibilität nach FO-BC Definition geprüft (Code Listing 8). Es werden nur Tokens akzeptiert, die Operatoren oder Prädikate der FO-BC Signatur darstellen und deren Anzahl an Argumenten zur Syntax passt.
Dieses Vorgehen wird nun mit jeder eingelesenen FO-BC Formel durch eine Schleife in der *main*-Funktion (Code Listing 9) wiederholt. Als Repräsentation für die weitere Verwendung wird ein Vektor mit Syntaxbäumen gefüllt. FO-BC Formeln, bei denen das Parsen aufgrund falscher Syntax fehlschlägt, werden für die weitere Berechnung verworfen und eine Warnmeldung wird ausgegeben. Um das Ergebnis des Parsings kontrollieren zu können wird eine Repräsentation des Syntaxbaums über die Systemkonsole ausgegeben, siehe Listing 3:

Listing 3: Konsolenausgabe

```
.....
SyntaxTree 5/5:
L-Branch of <=>
  L-Branch of &
    Mod with Arguments x and
  R-Branch of &
    A with Argument y
      L-Branch of =>
        Mod with Arguments y and
      R-Branch of =>
        TPO1 with Arguments x and y
R-Branch of <=>
  A with Argument z
    TPO2 with Arguments x and z




outputMode = random
reducedDiskSpace = 0



----------SUCESSFULLY FINISHED----------

Total States: 13
Anzahl gefundene einzelne Graphen: 22* 10^9
Execution time: 0s
```

5.1.3 Brute-Force Algorithmus

Die Umsetzung des gezeigten Kernalgorithmus aus Kapitel 5 erfolgt im Modul *belief_change_space.cpp*. Hierfür wir eine Klasse *changeOperator* eingeführt, siehe 10. Eine wichtige Frage bei der Implementierung ist die programminterne Darstellung des Graphen mit den Knoten, Kanten und zugewiesenen totalen Quasiordnungen zu jedem Knoten. Der Lösungsansatz besteht in der Verwendung von Vektoren. Ein Wissensoperator besitzt einen Vektor an Knoten, d.h. Zuständen. Für die Instanz eines Zustands wird eine Struktur *state* angelegt. Jeder Zustand beinhaltet einen Vektor an ganzzahligen Plausibilitätsindizes, die zusammen die mit dem Zustand verbundene totale Quasiordnung repräsentieren. Dabei entspricht die erste Zahl im Vektor dem Plausi-

bilitätsindex der Interpretation bzw. Welt *w0*, die zweite Zahl dem von *w1*, usw. Der Vektor besitzt also eine Länge, die der Anzahl an betrachteten Interpretationen entspricht. Die TPOs werden bereits beim Einlesen aus der Eingabedatei standardisiert (Code Listing 12), um sie mit der Liste an möglichen totalen Quasiordnungen vergleichen zu können.
Jeder Zustand besitzt außerdem einen weiteren Vektor bestehend aus den Kanten, die ihren Startpunkt in diesem Knoten haben. Eine Kante wird wiederum über eine neu definierte Struktur *edge* definiert. Diese besteht aus einer Ganzzahl, die den Index des Zielpunkts der Kante markiert. Außerdem besitzt jede Kante einen Vektor an Wahrheitswerten, der die Modelle der neuen Information der Kante repräsentiert. Dabei sagt der erste Wahrheitswert im Vektor aus, ob die erste betrachtete Welt *w0* Teil der Modelle der Kante ist, der zweite Wert, ob die zweite betrachtete Interpretation *w1* Teil der Modelle ist, usw. Auch dieser Vektor besitzt dadurch eine Länge, die der Anzahl an betrachteten Interpretationen entspricht.
Nachdem die Instanz der Klasse *changeOperator* mit den vordefinierten Zuständen und Kanten gefüllt ist, erfolgt die Vervollständigung, wie bereits vorgestellt, in zwei Schleifen mit Hilfe der Methode *complete* (siehe Code Listing 11):

- **Vervollständigung der fehlenden Zustände (Schleife 1)** Hierfür werden alle möglichen totalen Quasiordnungen erzeugt und mit denen der vordefinierten Zuständen zugewiesenen Ordnungen verglichen. Für alle Ordnungen, die noch nicht mit einem existierenden Zustand verknüpft sind, wird ein neuer Zustand eingefügt. Die Generierung der möglichen totalen Quasiordnungen ist auf zwei verschiedene Arten implementiert. Die erste Funktion gleicht einem Suchalgorithmus entlang einer Baumstruktur. Der Code dieser Funktion ist im Code Listing 13 zu finden. Es wird mit Hilfe dieser Funktion eine Tabelle erstellt, die alle möglichen totalen Quasiordnungen umfasst, auf der später der Abgleich zu bereits vorhandenen Zuständen erfolgt ist. Da die Anzahl an möglichen Ordnungen allerdings mit steigender Anzahl an betrachteten Welten schnell sehr groß wird, ist eine zweite Funktion mit verringertem Speicherbedarf implementiert. Diese startet bei Aufruf zwar das Suchverfahren neu, wodurch die Laufzeit gegenüber der Verwendung einer Tabelle verlängert wird, allerdings müssen dadurch nicht alle möglichen Quasiordnungen zusammen gespeichert werden. Für eine große Anzahl an betrachteten Interpretationen kann eine lange (evtl. auch mehrtägige) Laufzeit des Algorithmus in Kauf genommen werden, wohingegen ein zu großer Speicherbedarf die Ausführung verhindern kann. Zwischen den beiden Verfahren kann, durch Festlegung von *reducedDiskSpace* in der *belief_change_operator.h*-Datei, ausgewählt werden. Der Wert *true* führt zur Ausführung der Speicheroptimierten Funktion, der Wert *false* zum Aufruf der CPU-effizienteren Funktion.

- **Vervollständigung der fehlenden Kanten (Schleife 2)** Die Definition eines *vollständigen* Operators beinhaltet die Existenz einer Kante für jede mögliche Teilmenge der Menge von betrachteten Welten Ω. Hierfür erzeugt eine Funktion, zu sehen in Code Listing 14, alle möglichen Teilmengen, welche anschließend mit den vorhandenen Kanten abgeglichen werden. Es darf keine Kante für eine Teilmenge eingeführt werden, die bereits vordefiniert war, da der Änderungsraum nicht mehr deterministisch ist. Eine vordefinierte Kante darf auch nicht überschrieben werden, da das Ergebnis den Eingabeänderungsraum nicht mehr vollständig enthält.
  Für alle fehlenden Kanten, d.h. mit gegebenem Ausgangszustand und Teilmenge von Ω wird nun mit Hilfe einer Brute-Force-Funktion ein Zielzustand gesucht, der den Bedingungen aus den geparsten FO-BC Syntaxbäumen genügt. Dafür werden mehrere Schleifen ineinander verschachtelt:
  - **Schleife 3** iteriert über alle vorhandenen Zustände. Alle diese Zustände werden als potenzieller Zielzustand für die Kante überprüft. Für diese Schleife sind drei verschiedene Modi umgesetzt, welche über Festlegung des *outputMode* in der *belief_change_operator.h*-Datei gesteuert werden können. Im Modus *„single“* und *„random“* wird, sobald ein Zielzustand gefunden ist für den die FO-BC Bedingungen erfüllt sind, die Schleife abgebrochen und die gefundene Kante ergänzt. Im Modus *„all“* prüft der Algorithmus alle möglichen Kanten und erzeugt dadurch einen undeterministischen Änderungsraum. In diesem Modus erfolgt eine Ausgabe über die Systemkonsole, wie viele verschiedene vollständige Änderungsräume zur Eingabe gefunden wurden. Der Modus *„single“* unterscheidet sich vom Modus *„random“* in der Auswahl des nächsten zu prüfenden Zielzustandes für eine Kante. Während im Modus *„single“* die Zustände in chronologischer Reihenfolge geprüft werden, wird im Modus *„random“* der nächste zu prüfende Zielzustand zufällig ausgewählt. Dadurch lässt sich vermeiden, dass sich die Kanten vor allem auf wenige einzelne Knoten konzentrieren, sondern gleichmäßig verteilt werden. Falls die Schleife komplett durchlaufen ist, ohne dass ein Zielzustand ermittelt werden konnte, für den die Erfüllungsrelation der FO-BC Formeln gegeben ist, bricht der Algorithmus in allen drei Modi ab und liefert einen entsprechend negativen Rückgabewert. Auf Basis dieses Wertes wird durch die *main*-Funktion eine entsprechende Fehlermeldung ausgegeben, dass kein vollständiger Wissensoperator für die Eingaben existiert (siehe Listing Code Listing 15).
  - **Schleife 4** iteriert über alle möglichen Interpretationen der Konstante c. Für eine gültige Kante muss eine Interpretation für c gefunden werden, so

dass die FO-BC Bedingungen erfüllt sind. Diese Schleife bricht also auch bei der ersten gefundenen erfolgreichen Belegung ab. Innerhalb der Schleife wird nun jeweils auf eine Funktion zur Prüfung der Erfüllungsrelation zurückgegriffen. Um die Laufzeit im häufig zu erwartendem Sonderfall, dass die Konstante c kein Teil der FO-BC Formeln ist, zu verkürzen, wird für die zweite Schleife eine Prüfung vorgestellt. Eine zusätzliche Funktion prüft rekursiv alle Syntaxbäume auf Existenz der einzigen Konstante der FO-BC Signatur c (siehe Code Listing 17). Falls c kein Teil einer der Syntaxbäume ist, kann die Schleife entfallen und die Laufzeit verkürzt werden. Diese Verkürzung tritt bei negativen Ergebnissen der Prüfung der Erfüllungsrelation auf. Hier kann nach dem ersten negativen Ergebnis bereits der aktuelle Zielzustand verworfen werden, da eine Variation der Interpretation der nicht-existenten Konstanten c sich das Ergebnis nicht verändern wird.

5.1.4 Prüfung der Erfüllungsrelation

Die innerhalb der im letzten Abschnitt erläuterten Brute-Force-Schleifen aufgerufene Funktion zur Überprüfung der Erfüllungsrelation wird vom Modul *model_checking.cpp* zur Verfügung gestellt. Der darin definierte *Namespace* ist im Code Listing 16 zu sehen. Übergeben werden Ausgangszustand und Endzustand jeweils mit der zugeordneten Ordnung, die Modelle der neuen Information als Teilmenge von Ω sowie ein Vektor mit Referenzen auf die erstellten Syntaxbäume. Die Prüfung der Erfüllungsrelation erfolgt in weiteren verschachtelten Schleifen:

- **Schleife 5** iteriert über die Anzahl an Syntaxbäumen. Jede spezifizierte FO-BC Formel wird auf Erfüllung auf Basis des Syntaxbaums geprüft. Dabei muss die vorgeschlagene Kante alle FO-BC Formeln, d.h. Syntaxbäume erfüllen. Ein einziger nicht erfüllter Syntaxbaum führt zum Abbruch der Schleife, da die Erfüllungsrelation der Kante nicht mit allen FO-BC Formeln gegeben ist.
- **Schleife 6** iteriert über alle möglichen Variablenbelegungen von freien Variablen. Auch hier müssen wiederum alle Variablenbelegungen die FO-BC Formeln erfüllen, damit die Interpretation ein Modell der Bedingungen ist. Eine einzelne Variablenbelegung, die den gegebenen Syntaxbaum nicht erfüllt, führt zum Abbruch der Schleife und dem Ergebnis, dass die gegebene Kante den FO-BC Formeln nicht genügt, siehe Definition Wahrheitswert einer Formel in Kapitel 2.1.3. Für diese Schleife werden über eine Funktion zuerst alle freien Variablen gesucht und anschließend wird durch die möglichen Belegungen dieser iteriert.
- Im letzten Schritt wird rekursiv die Erfüllung des Syntaxbaums für jede der Va-

riablenbelegungen wahrheitsfunktional ausgewertet durch die Funktion *checkSubTree* (siehe Code Listing 18). Die Auswertung entspricht der im Kapitel 2.1.3 eingeführten Definition für die Auswertung des Wahrheitswerts einer Formel unter gegebener Variablenbelegung.

5.2 Laufzeitbetrachtungen

Die Anzahl der Schleifendurchläufe des Algorithmus aus 5 ist stark abhängig von der Anzahl der betrachteten Welten n.

Proposition 10 (Anzahl Knoten im vollständigen Änderungsraum) *Es sei O die Menge aller totalen Quasiordnungen im Eingabeänderungsraum und M die durch die vordefinierten Knoten beschriebene Multimenge über O, da für die Eingabe auch eine mehrfache Verwendung von Ordnungen zugelassen wird. Die Anzahl d an mehrfach verwendeten Ordnungen entspricht der Mächtigkeit der Menge der Elemente aus O, die in der Multimenge häufiger als einmal enthalten sind:*

$$d = \sum_{\{o \in O | M(o) > 1\}} M(o)$$

Die Anzahl N_{out} an Knoten im Ausgabeänderungsraum bei erfolgreichem Durchlauf des Algorithmus entspricht der Summe aus der Anzahl an möglichen totalen Quasiordnungen [50] und der Anzahl d an mehrfach verwendeten Ordnungen im Eingabeänderungsraum:

$$N_{out} = d + \sum_{k=0}^{n} k! \left\{ {n \atop k} \right\}$$

Der Ausdruck $\left\{ {n \atop k} \right\}$ steht dabei für die Stirling-Zahlen zweiter Art.
Ausgeschrieben berechnet sich N_{out} wie folgt:

$$N_{out} = d + \sum_{k=0}^{n} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n$$

$$= d + \sum_{k=0}^{n} \sum_{j=0}^{k} (-1)^{k-j} \frac{j^n k!}{j!(k-j)!}$$

Falls im Eingaberaum die Zustände alle eindeutig über ihre zugeordnete Ordnung bestimmt werden können, entfällt der erste Summand d, da der Algorithmus nur einzelne Zustände für fehlende Ordnungen einfügt.

Proposition 11 (Anzahl Kanten im vollständigen Änderungsraum) *Die Anzahl E_{out} an Knoten im Ausgabeänderungsraum bei erfolgreichem Durchlauf des Algorithmus ist das Produkt aus der Anzahl an möglichen Teilmengen von Ω und der Anzahl an Knoten im Ausgabeänderungsraum N_{out}:*

$$E_{out} = N_{out}(2^n - 1)$$

Über die Differenz der Anzahl an Knoten und Kanten im Eingabegraphen können die benötigten Schleifen des Algorithmus ermitteln werden:

Proposition 12 (Durchlauf Schleifen des Algorithmus) *Es sei N_{in} die Anzahl an Knoten im Eingabeänderungsraum. Die Schleife 1 des Algorithmus zum Einfügen von fehlenden Zuständen wird i_1 mal durchlaufen:*

$$i_1 = N_{out} - N_{in}$$

Es sei E_{in} die Anzahl an Kanten im Eingabeänderungsraum. Die Schleife 2 des erfolgreichen Algorithmus zum Einfügen von fehlenden Kanten wird i_2 mal durchlaufen:

$$i_2 = E_{out} - E_{in}$$

Die Durchläufe von Schleife 2 können geringer als hier angegeben sein, da der Algorithmus sofort abbricht, wenn für eine fehlende Kante kein passender Zielzustand ermittelt werden konnte. Deshalb ist in der Proposition vom „erfolgreichen Algorithmus“ die Rede.

Proposition 13 (Durchlauf Schleife 3 des Algorithmus) *Die innere Schleife 3 des erfolgreichen Algorithmus zur Prüfung von verschiedenen Zielzuständen wird pro zu suchender Kante durchschnittlich $\overline{i_3}$ mal durchlaufen, bis eine passende Kante gefunden wird. Dabei kann bereits beim ersten Durchlauf ein gültiger Zielzustand gefunden werden, oder alle möglichen Zielzustände durchprobiert werden. Das Intervall für $\overline{i_3}$ ist also wie folgt definiert:*

$$\overline{i_3} = [1, N_{out}]$$

Die innere Schleife 3 des erfolgreichen Algorithmus zur Prüfung von verschiedenen Zielzuständen wird insgesamt i_3 mal durchlaufen:

$$i_3 = i_2\overline{i_3}$$

Proposition 14 (Durchlauf Schleife 4 des Algorithmus) *Die innere Schleife 4 des erfolgreichen Algorithmus zur Prüfung von verschiedenen Interpretationen der Konstante c wird pro zu suchender Kante durchschnittlich $\overline{i_4}$ mal durchlaufen, bis eine passende Kante gefunden wird. Dabei kann bereits beim ersten Durchlauf ein gültiger Zielzustand gefunden werden, oder alle möglichen Belegungen von c durchprobiert werden. Das Intervall für $\overline{i_4}$ ist also wie folgt definiert:*

$$\overline{i_4} = [1, n]$$

Die innere Schleife 4 des erfolgreichen Algorithmus zur Prüfung von verschiedenen Interpretationen der Konstante c wird insgesamt i_4 mal durchlaufen:

$$i_4 = i_3\overline{i_4}$$

Dass $\overline{i_3}$ und $\overline{i_4}$ mindestens den Wert 1 besitzen, folgt aus der Bedingung, dass der Algorithmus erfolgreich ist, d.h. mind. eine Lösung für jede zu ergänzende Kante findet.

Proposition 15 (Durchlauf Schleife 4 ohne FO-BC Konstante) *Durch die vorherige Prüfung, ob die FO-BC Konstante c Teil einer der FO-BC Eingaben ist, gilt für Eingaben, die c nicht enthalten:*

$$\overline{i_4} = 1$$

Das bedeutet eine Verkürzung der Laufzeit um einen Faktor bis zu n.

Proposition 16 (Durchlauf Schleife 5 des Algorithmus) *Es sei S die Anzahl an eingegebenen FO-BC Formeln mit gültiger Syntax. Die innere Schleife 5 des erfolgreichen Algorithmus zur Prüfung der Erfüllungsrelation aller Syntaxbäume wird für Kanten, für die die Erfüllungsrelation nicht gegeben ist, durchschnittlich $\overline{i_5}$ mal durchlaufen, bis die Kante ausgeschlossen wird. Das Intervall für $\overline{i_5}$ ist also wie folgt definiert:*

$$\overline{i_5} = [1, S]$$

Die innere Schleife 5 des erfolgreichen Algorithmus zur Prüfung der Erfüllungsrelation aller Syntaxbäume wird insgesamt i_5 mal durchlaufen:

$$i_5 = \underbrace{\overline{i_5}(i_4 - i_2)}_{fehlgeschlagen} + \overbrace{i_2 S}^{erfolgreich}$$

Proposition 17 (Durchlauf Schleife 6 des Algorithmus) *Es sei V die Anzahl an freien Variablen in einer eingegebenen FO-BC Formel gültiger Syntax. Die innere Schleife 6 des erfolgreichen Algorithmus zur Prüfung der verschiedenen Variablenbelegungen wird für Kanten, für die die Erfüllungsrelation nicht gegeben ist, durchschnitt-*

lich $\overline{i_6}$ mal durchlaufen, bis die Kante ausgeschlossen wird. Das Intervall für $\overline{i_6}$ ist also wie folgt definiert:

$$\overline{i_6} = [1, \frac{1}{S} \sum_{m=1}^{S} n^{V_m}]$$

Die innere Schleife 5 des erfolgreichen Algorithmus zur Prüfung der verschiedenen Variablenbelegungen wird insgesamt i_6 mal durchlaufen:

$$i_6 = \underbrace{\overline{i_6}(i_5 - i_2)}_{fehlgeschlagen} + \overbrace{i_2 \sum_{m=1}^{S} n^{V_m}}^{erfolgreich}$$

Definition 21 (Erfolgs- und Fehlerfaktoren) *Die Faktoren der inneren Schleifen 3 ($\overline{i_3}$) und 4 ($\overline{i_4}$), welche ausdrücken wie viele Versuche durchschnittlich benötigt werden, um eine korrekte Kante zu finden, werden zusammengefasst zu einem gemeinsamen Faktor $\overline{i_{Success}}$:*

$$\overline{i_{Success}} = \overline{i_3 i_4}$$

Die Faktoren der inneren Schleifen 5 ($\overline{i_5}$) und 6 ($\overline{i_6}$), welche Ausdrücken wie viele Versuche durchschnittlich benötigt werden, um ungültige Kanten zu identifizieren, werden zusammengefasst zu einem gemeinsamen Faktor $\overline{i_{Fail}}$:

$$\overline{i_{Fail}} = \overline{i_5 i_6}$$

Proposition 18 (Ausführung rekursiver Erfüllungscheck) *Die Anzahl an initialen Aufrufen der Funktion für rekursiven Erfüllungscheck checkSubTree (Listing 18) zur Prüfungen der Erfüllungsrelation auf einem Syntaxbaum i_{total}, die der Algorithmus bei gegebener Eingabe durchführt, kann über folgende Formel ausgedrückt werden:*

$$i_{total} = i_6 = i_2(\sum_{m=1}^{S} n^{V_m} + \overline{i_{Fail}}(\overline{i_{Success}} - 1))$$

$$= \left[(2^n - 1)\left(\sum_{k=0}^{n} k! \left\{ {n \atop k} \right\} + d\right) - E_{in}\right](\sum_{m=1}^{S} n^{V_m} + \underbrace{\overline{i_{Fail}}(\overline{i_{Success}} - 1)}_{P})$$

Bei gegebenen FO-BC Formeln und gegebenem Eingangsänderungsraum, d.h. bekanntem d, n, E_{in}, V und S hängt die Laufzeit also weiterhin von den beiden Faktoren $\overline{i_{Fail}}$ und $\overline{i_{Success}}$ ab. Im besten Fall, wenn in den Schleifen 3 und 4 jeweils beim ersten Versuch eine korrekte Kante ausgewählt wird, reduziert sich die letzte Differenz P und somit das letzte Produkt bzw. der letzte Summand in der Formel zu 0.

Proposition 19 (Grenzbetrachtung Prüfung Erfüllungsrelation) *Die minimalen Durchläufe der innersten Schleife des Algorithmus, d.h. die Anzahl an minimal durchzuführenden Prüfungen der Erfüllungsrelation beträgt:*

$$\underset{i_{Fail},i_{Success}}{\min}(i_{total}) = \left[(2^n - 1)\left(\sum_{k=0}^{n} k!\left\{{n \atop k}\right\} + d\right) - E_{in}\right] \sum_{m=1}^{S} n^{V_m}$$

Die maximalen Durchläufe ergeben sich zu:

$$\underset{i_{Fail},i_{Success}}{\max}(i_{total}) = \left[(2^n - 1)\left(\sum_{k=0}^{n} k!\left\{{n \atop k}\right\} + d\right) - E_{in}\right] N_{out} n \sum_{m=1}^{S} n^{V_m}$$

Die Differenz an Prüfungen der Erfüllungsrelation, die zwischen einer optimalen Auswahl der Kanten und der schlechtesten möglichen Auswahl an Kanten besteht, beträgt also:

$$\Delta_{i_{total}} = \left[(2^n - 1)\left(\sum_{k=0}^{n} k!\left\{{n \atop k}\right\} + d\right) - E_{in}\right] (N_{out} n - 1) \sum_{m=1}^{S} n^{V_m}$$

Abbildung 9 visualisiert die starke Abhängigkeit der benötigten initialen Aufrufe des rekursiven Erfüllungschecks *checkSubTree* (Listing 18) im Algorithmus von der Anzahl an betrachteten Welten. Zur Vereinfachung sind in der Abbildung die Funktionen für eine Eingabe ohne vordefinierte Kanten und Eingabe einer einzelnen FO-BC Formel geplottet.

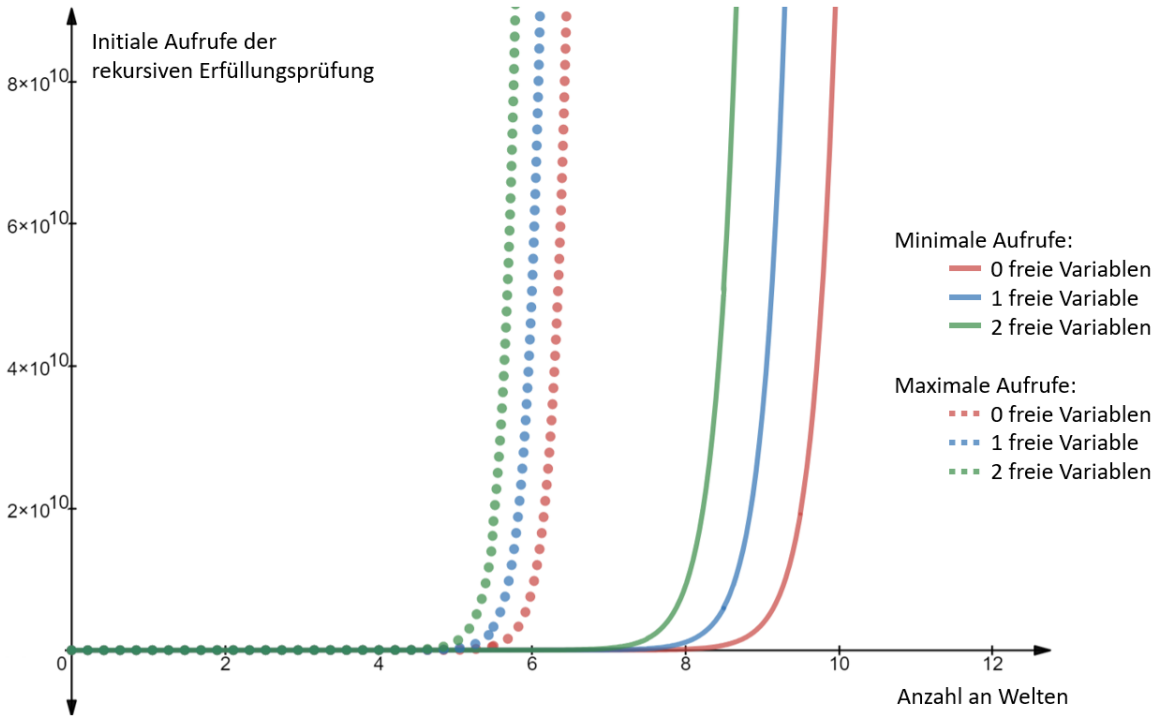

Abbildung 9: Zusammenhang von Laufzeit zu Welten und freien Variablen

Tabelle 6 sind, für die getroffene Vereinfachung aus Abbildung 9, die minimalen und maximalen Aufrufe des rekursiven Erfüllungschecks, ohne freie Variable in der FO-BC Formel, als Zahlenwerte zu entnehmen.

Anzahl Prädikatensymbole	Anzahl Welten n	minimale Aufrufe	maximale Aufrufe
1	2	9	54
2	4	$1,1 * 10^3$	$3,4 * 10^5$
3	8	$1,4 * 10^8$	$6,1 * 10^{14}$
4	16	$3,5 * 10^{20}$	$3,0 * 10^{37}$

Tabelle 6: Initiale Aufrufe des rekursiven Erfüllungschecks

Der eingeführte Summand P kann als Perfomance Indikator für das Auswahlverfahren der nächsten zu prüfenden Kante des Brute-Force Algorithmus herangezogen werden:

Proposition 20 (Performance Indikator für Kantenauswahl) *Um eine einfache Kennzahl zum Vergleich zwischen zwei Verfahren zur Auswahl der nächsten zu prüfenden Kante für den Algorithmus zu erhalten, kann der zweite Summand der Gleichung zur Bestimmung der initialen Ausführungen der rekursiven Prüfung der Erfüllungsrelation P herangezogen werden:*

$$P = \overline{i_{Fail}}(\overline{i_{Success}} - 1)$$

Ein Wert $P = 0$ stellt damit die schnellstmögliche Berechnung des Änderungsraumes dar, während bei steigendem P häufig ungültige Kanten überprüft werden, bevor eine gültige Kante ermittelt werden kann. Falls $\overline{i_{Fail}}$ und $\overline{i_{Success}}$ nicht bekannt sind, aber die Gesamtanzahl an initialen Aufrufen der *checkSubTree*-Funktion, kann P durch Umstellen der Formel aus Proposition 18 berechnet werden.

5.3 Prämissen zur Vervollständigung

Für die Implementierung sind Prämissen getroffen. Diese sind im Folgenden mit Betrachtungen zu ihrer Auswirkung aufgelistet:

- **Betrachtete Teilmengen von Ω zur Vervollständigung der Kanten**: Der Algorithmus fügt in der zweiten Schleife ab Zeile 6 die fehlenden Kanten des Graphen ein. Dafür wird eine Kante für jede Kombination aus Ausgangszustand $s \in S^*$ und Modellen einer neuer Information $\alpha \subseteq \Omega$ gesucht. Für die Umsetzung wird definiert, dass die leere Menge $\alpha = \emptyset$ keine Kante im Graphen erhält. Die leere Menge entspricht einer neuen Information, die keine Modelle (innerhalb

der betrachteten Interpretationen) besitzt. Dieses Verhalten ist intuitiv nachvollziehbar: Eine neu erhaltene Information eines Agenten, die durch keine der betrachteten Welten erfüllt wird, führt durch die getroffene Festlegung zu keiner Änderung der Überzeugung des Agenten. Beispiel hierfür ist der Erhalt einer widersprüchlichen Information.

- **Kanten mit demselben Ausgangs- und Endpunkt**: Eine weitere wichtige Entscheidung muss bezüglich der möglichen Einschränkung der einzufügenden Kanten getroffen werden. Prinzipiell könnte definiert werden, dass sich der Zustand nach einer Überzeugungsänderung vom Zustand vorher unterscheiden muss, d.h. $s^* \neq s$. Auch wenn die Einschränkung vermeintlich intuitiv korrekt erscheinen mag, führt sie zu kontraintuitiven Resultaten. Denn eine Kante im Graphen, d.h. Überzeugungsänderung, ist formal durch die Kombination eines Ausgangszustand mit einer neuen Information definiert. Der Begriff der Überzeugungsänderung ist an der Stelle allerdings irreführend, da auch der nochmalige Erhalt einer bereits in der Vergangenheit erhaltenen Information eine solche Kante definiert, siehe Beispiel 10.

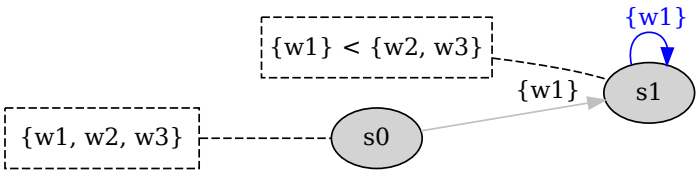

Abbildung 10: Beispiel: Kanten mit demselben Ausgangs- und Endpunkt

Durch Einführung dieser Einschränkung würden einige FO-BC Postulate nicht mehr erfüllbar. Als Beispiel ist das in Kapitel 4 eingeführte Postulat CR1 zu nennen. Deshalb wird die Einschränkung nicht getroffen und alle $s \in S^*$ als mögliche Endpunkte von Kanten betrachtet, d.h. explizit auch $s^* = s$.

- **Zustände mit derselben Quasiordnung** Für den Algorithmus sind „doppelte“ Zustände mit derselben zugeordneten totalen Quasiordnung für den Eingangsoperator zugelassen. Während der Vervollständigung der Zustände im Durchlauf des Algorithmus werden hingegen keine Zustände mit bereits zugewiesenen Ordnungen erstellt. Diese Erweiterung wäre sinnvoll und ggf. notwendig, wenn die Bedingungen in einer mächtigeren Sprache als dem FO-BC Fragment formuliert würden. Wenn beispielsweise Bedingungen nicht nur für die Ordnungen der zwei Zustände einer Kante formuliert werden, sondern zusätzlich die Ordnungen der möglichen vorherigen Zustände berücksichtigt werden, kann das Einfügen von mehreren Zuständen mit gleicher zugewiesener Ordnung notwendig sein. Für die hier verwendete FO-BC Signatur kann das Einfügen dieser „doppelten“ Zustände allerdings entfallen.

- **Betrachtete Ordnungen** Gemäß der Definition Änderungsraum aus Kapitel 3 werden für die den Zuständen zugewiesenen Ordnungen ausschließlich totale Quasiordnungen betrachtet. Wie der Autor in [43] belegt, existieren auch solche Operatoren, die beispielsweise den AMG Postulaten (G*1) - (G*6) genügen, aber auf partiellen Ordnungen basieren. Diese Operatoren können durch die eingeführte Definition von Änderungsräumen nicht dargestellt werden.

5.4 Beispiele

In diesem Kapitel werden einige Beispiele für Belief-Change-Operatoren vorgestellt, welche mit Hilfe des Programmes erstellt werden. Dabei kann je nach Auswahl der Postulate ein Revisions- oder Kontraktionsoperator erzeugt werden.

5.4.1 Modellierung AGM-Postulate für Revision

Um einen AGM-Revisionsoperator zu erzeugen, wird das Repräsentationstheorem für die Revision von Katsuno und Mendelzon, das in Kapitel 2.3 eingeführt wird, verwendet. Demnach muss eine Zuordnung zu einer totale Quasiordnung existieren, die den Bedingungen für ein *faithful assignment* genügt. Das wird dadurch sichergestellt, dass für die Beispiele ausschließlich die Interpretationen (Welten) als Modelle bezeichnet werden, welche den Plausibilitätsrang 0 in der Quasiordnung besitzen. Dass die eingegebenen Ordnungen totale Ordnungen darstellen, muss nicht geprüft werden, da das Programm ausschließlich mit totalen Präordnungen arbeitet. Außerdem müssen nach dem Repräsentationstheorem die Modelle, d.h. die Welten mit Plausibilitätsrang 0, für die Revision als $Mod(\psi \circ \mu) = Min(Mod(\mu), \leq_\psi)$ berechnet werden. Die Bedingung muss als FO-BC Postulat für die Eingabe in das Programm formuliert werden:

$$(Mod(x)\&(A(y)(Mod(y) => TPO1(x,y)))) <=> (A(z)TPO2(x,z))$$

Mit Hilfe dieser Eingabe und der Festlegung der Interpretationen mit kleinstem Rang bzgl. der totalen Quasiordnung kann nun ein AGM-Revisionsoperator berechnet werden. Mit Hilfe der Wahl des Ausgabemodus *outputMode = „all“* kann eine Größenordnung für die Anzahl an gefundenen vollständigen Revisionsoperatoren erhalten werden. Für die erste Ausführung wird eine Anzahl von vier Welten festgelegt. Das entspricht der Anzahl an möglichen Interpretationen, die für eine Sprache über zwei aussagenlogischen Symbolen existieren. Für eine leere Eingabe des Änderungsraums, d.h. keine vordefinierten Zustände oder Kanten, werden vom Programm $14 * 10^{1125}$ mögliche Alternativen für einen vollständigen AGM-Revisionsoperator gefunden. Ein solcher einzelner Graph, welcher mit dem Ausgabemodus *outputMode = „single“* oder *outputMode = „random“* erzeugt werden kann, besitzt bereits 75 Zustände und 1125 Kanten. Dadurch ist eine Visualisierung bereits für vier betrachtete Welten kaum sinnvoll möglich.

In einer zweiten Ausführung wird eine Anzahl von lediglich zwei Welten festgelegt, bei ebenfalls leerem Eingabe-Änderungsraum. Das entspricht einem einzigen aussagenlogischen Symbol, das entweder *WAHR* oder *FALSCH* sein kann. Dafür existiert nur noch ein einziger vollständiger Operator mit drei Zuständen und neun Kanten. Abbildung 11 zeigt den aus der vom Programm erstellten *.dot*-Datei erzeugten Graphen. Dazu wird die erzeugte *.dot*-Datei in Abbildung 11 von einem *GraphViz*-Renderer interpretiert.

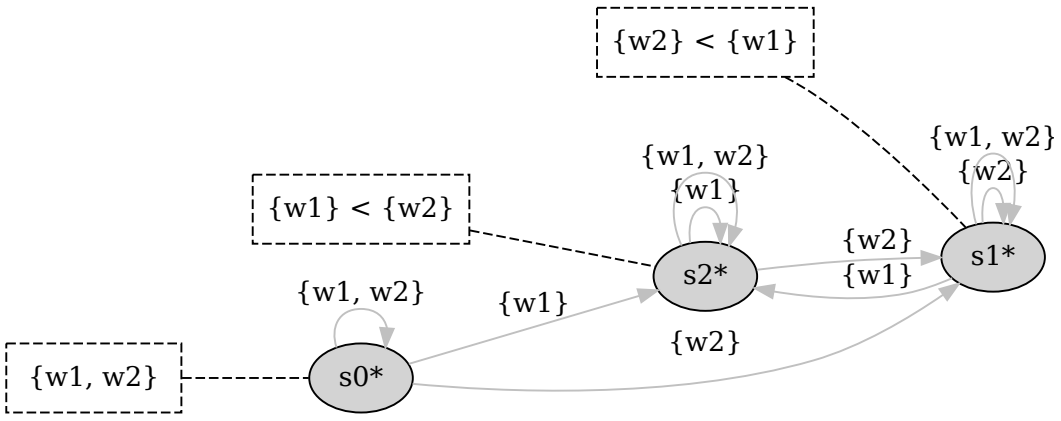

Abbildung 11: AGM-Revisionsoperator mit zwei Welten

5.4.2 Modellierung Postulate von Darwiche und Pearl

Die ergänzenden Postulate von Darwiche und Pearl für iterierte Revision, die im Kapitel 2.4 vorgestellt werden, sollen nun ebenfalls verwendet werden, um einen Revisionsoperator zu generieren. Neben der FO-BC Bedingung für die AGM-Postulate aus dem letzten Kapitel und der Festlegung der Interpretationen mit kleinstem Plausibilitätsrang als Modelle, werden die Postulate (CR1) - (CR4) in weitere FO-BC Formeln überführt und dem Programm als Eingabe übergeben:

(CR1_FOBC) $(Mod(x)\&Mod(y)) => (TPO1(x,y) <=> TPO2(x,y))$
(CR2_FOBC) $(\sim Mod(x)\& \sim Mod(y)) => (TPO1(x,y) <=> TPO2(x,y))$
(CR3_FOBC) $(Mod(x)\& \sim Mod(y)) => (\ TPO1(y,x) =>\ TPO2(y,x))$
(CR4_FOBC) $(Mod(x)\& \sim Mod(y)) => (TPO1(x,y) => TPO2(x,y))$

In einer ersten Ausführung wird erneut eine Anzahl von vier Welten festgelegt, bei leerem Eingabe-Änderungsraum. Die Anzahl an gefundenen vollständigen Änderungsräumen reduziert sich gegenüber der alleinigen Angabe der AGM-Bedingungen auf $21 * 10^{311}$ mögliche Alternativen, mit wieder je 75 Zuständen und 1125 Kanten. Da hier die Visualisierung wieder nicht möglich ist, wird in einer zweiten Durchführung die Anzahl an Welten auf zwei reduziert.

In der Ausführung mit zwei Welten wird der gleiche einzige vollständige Änderungsraum aus Abbildung 11 gefunden, wie er bereits nur durch Angabe der AGM-Revisionsbedingungen erzeugt wird. Die ergänzenden Postulate (CR1) - (CR4) von Darwiche und Pearl widersprechen also nicht dem bereits gefunden einzigen AGM-Änderungsraum auf zwei Welten.

5.4.3 Modellierung AGM-Postulate für Kontraktion

In Kapitel 2.3 wird gezeigt, dass ein AGM-Revisionsoperator genutzt werden kann, um über die Levi-Identität einen AGM-Kontraktionsoperator zu definieren. So kann auch mit den in den letzten beiden Kapiteln erzeugten Revisionsoperatoren vorgegangen werden. Allerdings kann das Programm auch genutzt werden, um direkt einen AGM-Kontraktionsoperator zu erzeugen. Dafür wird das Repräsentationstheorem für die Kontraktion von Caridroit et. al., das in Kapitel 2.3 eingeführt wird, verwendet. Analog zum Repräsentationstheorem für die Revision wird ein *faithful assignment* benötigt, welches dadurch erfüllt wird, dass im Programm die Welten mit niedrigstem Plausibilitätsrang (d.h. 0) als Modelle gelten. Das Programm arbeitet ausschließlich mit totalen Quasiordnungen, wodurch auch diese Bedingung erfüllt ist. Außerdem müssen nach dem Repräsentationstheorem die Modelle, d.h. die Welten mit Plausibilitätsrang 0, für die Kontraktion als $Mod(\psi - \mu) = Mod(\psi) \cup Min(Mod(\neg\mu), \leq_\psi)$ berechnet werden. Die Bedingung muss als FO-BC Postulat für die Eingabe in das Programm formuliert werden:

$$((\sim Mod(x)\&(A(y)(\sim Mod(y) => TPO1(x,y))))|(A(y)TPO1(x,y))) \ldots$$

$$\cdots <=> (A(z)TPO2(x,z))$$

Mit Hilfe dieser Eingabe und der Festlegung der Interpretationen mit kleinstem Rang bzgl. der totalen Quasiordnung kann nun ein AGM-Kontraktionsoperator berechnet werden. Mit Hilfe der Wahl des Ausgabemodus *outputMode = „all“* kann eine Größenordnung für die Anzahl an gefundenen vollständigen Revisionsoperatoren erhalten werden.
Für die erste Ausführung wird eine Anzahl von vier Welten festgelegt. Das entspricht der Anzahl an möglichen Interpretationen, die für eine Sprache über zwei aussagenlogischen Symbolen existieren. Für eine leere Eingabe des Änderungsraums, d.h. keine vordefinierten Zustände oder Kanten, werden vom Programm $12 * 10^{708}$ mögliche Alternativen für einen vollständigen AGM-Kontraktionsoperator gefunden. Ein solcher einzelner Graph, welcher mit dem Ausgabemodus *outputMode = „single“* oder *outputMode = „random“* erzeugt werden kann, besitzt wieder 75 Zustände und 1125 Kanten. Dadurch ist eine Visualisierung bereits für vier betrachtete Welten kaum sinnvoll möglich.

In einer zweiten Ausführung wird eine Anzahl von lediglich zwei Welten festgelegt, bei ebenfalls leerem Eingabe-Änderungsraum. Das entspricht einem einzigen aussagenlogischen Symbol, das entweder *WAHR* oder *FALSCH* sein kann. Dafür existiert nur noch ein einziger vollständiger Operator mit 3 Zuständen und 9 Kanten. Abbildung 11 zeigt den aus der vom Programm erstellten *.dot*-Datei erzeugten Graphen. Dazu wird die erzeugte *.dot*-Datei von einem *GraphViz*-Renderer interpretiert:

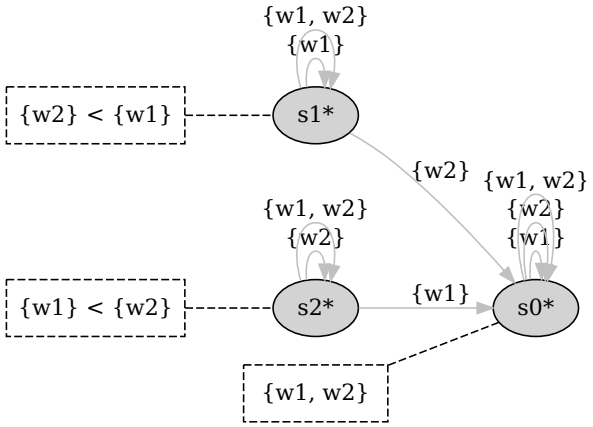

Abbildung 12: AGM-Kontraktionsoperator mit zwei Welten

6 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde ein Programm zur Vervollständigung von partiellen Änderungsräumen, welche Repräsentationen von Änderungsoperatoren darstellen, erstellt und untersucht. Die bei der Vervollständigung eingesetzte Brute-Force-Methodik führt zu steil ansteigender Rechenzeit für eine größere Anzahl von betrachteten Interpretationen. Dabei steigt die Anzahl an möglichen Interpretationen über die Anzahl an aussagenlogischen Symbolen der zu Grunde liegenden Sprache bereits exponentiell ($O(2^n)$). Daraus resultiert bereits für zwei betrachtete aussagenlogische Symbole, d.h. vier betrachtete Welten, ein Änderungsraum, der durch die hohe Anzahl an Kanten im Graphen schwierig zu visualisieren ist. Um das erstellte Programm auch für größere Änderungsräume verwenden zu können, wurde der Algorithmus möglichst effizient durch beispielsweise den Einsatz von Referenzen und in kompilierter Sprache (C++) programmiert. Um diese Skalierbarkeit zu gewährleisten, stellt auch der Speicherbedarf des Programmes eine Begrenzung dar. Da dieser im Gegensatz zur Rechenzeit immer physikalisch begrenzt ist, wurden speicherintensive Teile umgeschrieben, so dass eine längere Laufzeit für Funktionen einem erhöhten Speicherbedarf vorgezogen wird. Da die Anzahl an Welten den größten Einfluss auf die Ressourcennutzung hat, sollten immer nur für den Anwendungsfall tatsächlich notwendige Welten betrachtet werden.

Die Implementierung stellt eine Grundlage für weitere Untersuchungen dar. Mit Hilfe der eingeführten Kennzahl P können verschiedene Verfahren zur Auswahl der nächsten zu prüfenden Kante miteinander verglichen werden. Dadurch könnte für bestimmte Sonderfälle die Berechnung des vollständigen Änderungsraums weiter beschleunigt werden. Um P weiter zu verallgemeinern, könnte P um die Anzahl an möglichen Kanten in dem gesuchten Änderungsraum normalisiert werden. Außerdem besteht durch die zusätzlich umgesetzten Ausgabemodi auch die Möglichkeit, sich alle möglichen vollständigen Änderungsräume zu gegebener Eingabe berechnen zu lassen, um so die absolute Anzahl der existierenden Alternativen und bestimmte Alternativen zu vergleichen. Zur praktischen Anwendung der vervollständigten Operatoren wird ein weiteres Programm benötigt, welche bei gegebenem Ausgangszustand und neuer Information das Resultat der Operation im Graphen bestimmt. Für diese Ausführung des Operators besteht eine weitere Möglichkeit neben der naheliegenden Verwendung eines einzelnen vollständigen Änderungsraums. Dabei können alle möglichen vollständigen Wissensänderungsoperatoren zu einer Eingabe berechnet werden. In der Anwendung könnte nicht nur das Ergebnis der Operation über einen erzeugten Graphen bestimmt werden, sondern es könnten alle gefunden Graphen einbezogen werden. Eine Möglichkeit wäre die Vereinigung der Modelle aller Zielzustände aus sämtlichen möglichen vollständigen Operatoren zu bilden. Eine andere Möglichkeit besteht beispielsweise darin über ein Ranking System den Rang der Welten über alle möglichen Zielzustände zu addieren

und so eine neue Ordnung über den Welten zu generieren. Als alternative Darstellung zu einer Sammlung von möglichen Änderungsräumen, können alle gefundenen Kanten auch in einen dann nicht-deterministischen Graphen zusammengefasst werden, auf dem ein solches System implementiert werden kann. Dieser Graph ist Ausgabe des umgesetzten Ausgabemodus „*all*“.

Für das Programm sind weitere Optimierungen und Erweiterungen denkbar. Für weitere Laufzeitverbesserungen kann die Überprüfung der Erfüllungsrelation durch eine effizientere Methode aus der Literatur umgesetzt werden. FO-BC Formeln könnten durch die Umformung in Normalformeln und z.B. die Erkennung von Tautologien und elementaren Widersprüchen effizienter geprüft werden. Des Weiteren können die getroffenen Prämissen für die Umsetzung erweitert werden. Der Algorithmus wurde für die Eingabe von Änderungsräumen mit totalen Quasiordnungen definiert. Allerdings lassen sich dadurch ein großer Teil von Wissensänderungsoperatoren nicht darstellen, obwohl beispielsweise auch AGM-Operatoren existieren, die über eine partielle Ordnung repräsentiert werden. Neben der Ordnung könnte die Sprache zur Definition der Bedingungen der Überzeugungsänderungen, die FO-BC, erweitert werden. Der von Darwiche und Pearl erwähnte den AGM-Revisionsoperator von Spohn, der die Postulate (CR1) - (CR4) erfüllt und auf einem Ranking der Welten basiert, ist zwar als Änderungsraum darstellbar, kann aber durch Angabe von FO-BC Formeln nicht ausgedrückt und dadurch nicht berechnet werden. Eine Möglichkeit die berechenbaren Operatoren zu erweitern besteht darin, dass nicht die Welten kleinster Ordnung als Modelle des Ergebnisses der Operation definiert werden, sondern eine separate Zuordnung zwischen der ersten über FO-BC Formeln definierten Ordnung zu den Modellen über zusätzliche Formeln existiert. Dadurch ist jedem Zustand eine zweite Ordnung zugewiesen, mit der komplexere Bedingungen als in aktueller FO-BC Sprache formuliert werden können. Für den eingegebenen Änderungsraum könnten außerdem vordefinierte Kanten auf Erfüllung der FO-BC Formeln geprüft werden, um die Einschränkung der Eingabe auf Änderungsräume, die die FO-BC bereits erfüllen, aufzuheben.

7 Code Listings

Listing 4: Der Namespace fileproc::

```
namespace fileproc {

  //----------------------input-----------------------------//

  struct rawEdge {
    changeOperator::state::edge edge;
    int origin = 0;
  };

  struct rawState {
    vector<int> TPO;
    string name;
  };

  vector<string> read_FOBC_input(void);

  vector<rawState> get_predefStates(vector<string>& wnames);

  vector<rawEdge> get_predefEdges(const vector<string>& wnames, const
      vector<fileproc::rawState>& states);

  .....

  //----------------------output----------------------------//

  int output_completedOperator(const changeOperator& CO, const vector<
      fileproc::rawEdge>& pdEs, const vector<fileproc::rawState>& pdSs,
      const vector<string>& wnames);

  .....

}
```

Listing 5: Klasse SyntaxTree

```
class SyntaxTree {

  public:

    struct node {
      node* parent, * left, * right;
      string type;
      vector<string> args;

```

```
10        node() : parent(nullptr), left(nullptr), right(nullptr), type(""),
              args(2, "") {}
11      };
12
13      node* root;
14
15      SyntaxTree() {
16        this->root = nullptr;
17      }
18
19      ~SyntaxTree() {
20        deleteBranch(this->root);
21      }
22
23      int parse_FOBC(const string &s);
24
25      static bool isQuantor(const string& s);
26
27      //output syntax tree to system console (debugging)
28      static void createExpressionString(const node* root, const int& n) ;
29
30    private:
31
32      static node* createNode(const string& s);
33
34      static bool isOperator(const string& s);
35
36      static bool isValidArg(const string& arg, const int& argcount, const
            string& name);
37
38      static int operatorPrecedence(const string& op);
39
40      static node* createSyntaxTree(const vector<string>& expression);
41
42      static vector<string> tokenize(const string& s_in);
43
44      static void deleteBranch(node* root);
45
46      static int addToStack(const string& op, stack<node*>& operandStack)
            {....}
47
48    };
```

Listing 6: FOBC Parser Methode

```
int SyntaxTree::parse_FOBC (const string& s)
//parse FOBC to SyntaxTree
{
    this->root = createSyntaxTree(tokenize(s));
    if (this->root == nullptr)
      { return 1; }
    return 0;
}
```

Listing 7: Bindungsprioritäten der Operatoren

```
int SyntaxTree::operatorPrecedence(const string& op)
//get operator precedences as rank integer from name
{
  /*if (op == "(" || op == ")")
  return 8;*/
  if (op == "~")
    return 7;
  if (op == "&")
    return 6;
  if (op == "|")
    return 5;
  if (op == "=>")
    return 4;
  if (op == "<=>")
    return 3;
  if (op == "A")
    return 2;
  if (op == "E")
    return 1;
  return 0;
}
```

Listing 8: Prüfung auf valide FOBC Tokens

```
bool SyntaxTree::isValidArg(const string& arg, const int& argcount, const
    string& name)
//check if input string arg as argument nr argcount is valid for node
   with type name
{
  if (arg[0] == 'c' || arg[0] == 'u' || arg[0] == 'v' || arg[0] == 'w'
    || arg[0] == 'x' || arg[0] == 'y' || arg[0] == 'z') { //check for
       allowed variables and constants
    bool numeric_args = true;
    for (auto i = 1; i < arg.length(); i++) { //numbers after letters are
        allowed to increase possible variables to use
      if (!isdigit(arg[i])) {
        numeric_args = false;
```

```
10        }
11      }
12      if (numeric_args && arg.length() <= 3) { //only two numbers are
            allowed to index variables
13
14        if ((name == "Mod" || isQuantor(name)) && argcount < 1) //predicate
              "Mod" and quantors have only one argument
15          return true;
16
17        if ((name == "TPO1" || name == "TPO2") && argcount < 2) //
              predicates "TPO1" and "TPO2" have two arguments
18          return true;
19      }
20    }
21    return false;
22  }
```

Listing 9: Parsen der FOBC Formeln in main()

```
1   vector<string> FOBC_input = fileproc::read_FOBC_input(); // read FOBC
        input from .txt file
2   vector<SyntaxTree> FOBC_trees(FOBC_input.size());;
3
4   for (auto i = 0; i < FOBC_trees.size(); i++) {
5     switch (FOBC_trees[i].parse_FOBC(FOBC_input[i])) { // print syntax
          tree or error (debugging)
6       case 0:
7         cout << "SyntaxTree " << i+1 << "/" << FOBC_trees.size() << ":\n"
              ;
8         SyntaxTree::createExpressionString(FOBC_trees[i].root, 0);
9         cout << endl << endl;
10        break;
11      case 1:
12        cout << "The following FOBC input has been ignored due to invalid
              syntax:\n" << FOBC_input[i] << endl << endl;
13    }
14  }
```

Listing 10: Klasse changeOperator

```
1  class changeOperator {
2
3  public:
4
5    changeOperator(const int& world_count) {
6      this->worlds = world_count;
7      graphs = 1;
8      graphs_pow = 0;
9    }
```

```
10
11    void addState(const vector<int>& TPO);
12
13    void addEdge(const int& orig, const int& dest, const vector<bool>&
          alpha);
14
15    int complete(const vector<SyntaxTree>& FOBC_Trees);
16
17    struct state {
18
19      struct edge {
20        vector<bool> alpha;
21        int dest;
22
23        edge() : alpha(), dest() {}
24        edge(const int& dest_in, const vector<bool>& alpha_in) : alpha(
              alpha_in), dest(dest_in) {}
25      };
26
27      vector<int> TPO;
28      vector<edge> edges;
29
30      state() : TPO(), edges() {}
31      state(const vector<int>& TPO_in) : TPO(TPO_in), edges() {}
32
33    };
34
35    vector <state> states;
36    int graphs;
37    int graphs_pow;
38
39    void debugOutput(void) { ... } //output change operator to system
          console (debugging)
40
41  private:
42
43    vector<vector<int>> TPOtable;
44    int worlds;
45
46    void genTPOtable(const vector<int>& root_TPO);
47
48    bool genTPOs_v2(int& index, vector<int>& TPO);
49
50    vector<int> getTPO(const int& index);
51
52    bool checkTPO(const vector<int>& TPO);
53
54    vector<bool> generateSubsets(const int& index);
```

```
55
56   void completeStates(void);
57
58   int completeEdges(const vector<SyntaxTree>& FOBC_cond);
59
60   int bruteForce(const int& origState, const vector<bool>& alpha, const
         vector<SyntaxTree>& FOBC_cond);
61
62   int poweroften(int x);
63
64 };
```

Listing 11: Methode zur Vervollständigung der Wissensoperatoren

```
1 int changeOperator::complete(const vector<SyntaxTree>& FOBC_Trees)
2 //complete states and edges of this operator object
3 {
4   completeStates(); //loop 1 for adding missing states
5   cout << "\n\noutputMode = " << outputMode << endl;
6   cout << "reducedDiskSpace = " << reducedDiskSpace << endl << endl;
7   return (completeEdges(FOBC_Trees)); //loop 2 for adding missing edges
8 }
```

Listing 12: Standardisierung der TPOs

```
1 vector<int> fileproc::normalizeTPO(const vector<int>& input)
2 //normalize input TPO
3 {
4   int max_order = input.size() - 1;
5   vector<int> output = input;
6   for (auto i = 0; i < max_order; i++) {
7     if (count(output.begin(), output.end(), i) == 0) {
8       for (auto n = 0; n < output.size(); n++) {
9         if (output[n] > i)
10          output[n] = output[n] - 1;
11      }
12    }
13  }
14  return (output);
15 }
```

Listing 13: Generierung von totalen Quasiordnungen (CPU optimiert)

```
void changeOperator::genTPOtable(const vector<int>& root_TPO)
//generate table for this object containing all possible TPOs depending
    on the nr. of worlds
//tree-like search pattern (cpu efficient, not memory efficient)
{
  int max_order = this->worlds - 1;
  for (auto i = 0; i < this->worlds; i++) {
    if ((root_TPO[i] < max_order) && (count(root_TPO.begin(), root_TPO.
        end(), root_TPO[i]) > 1)) {
      vector<int> mod_TPO = root_TPO;
      mod_TPO[i]++;
      if (count(this->TPOtable.begin(), this->TPOtable.end(), mod_TPO) ==
          0) {
        this->TPOtable.push_back(mod_TPO);
        genTPOtable(mod_TPO);
      }
    }
  }
  return;
}
```

Listing 14: Generierung von Teilmengen

```
vector<bool> changeOperator::generateSubsets(const int& index)
//creates possible subsets of nr. of worlds of object (vector of bools)
{
  int num = index;
  vector<bool> output(this->worlds, 0);
  for (auto i = 0; i < this->worlds; i++) {
    output[i]= num % 2;
    num = num / 2;
    if (num == 0)
      break;
  }

  return output;
}
```

Listing 15: Vervollständigung des Wissensoperators in main()

```
  switch (CO.complete(FOBC_trees)) { //complete operator and check result

    case 1: //failure
      cout << "\n\n-------------------------COMPLETION FAILED
          -------------------------\n\n";
      cout << "There exists no complete operator for your input.\n";
      cin.get();//leave console open
      return 1;
```

```
8
9       case 0: //success
10
11          //CO.debugOutput(); //visualize completed operator in system
                console
12
13          //output summary in system console
14          cout << "\n\n-------------------------SUCESSFULLY FINISHED
                -------------------------\n\n";
15          cout << "Total States: " << CO.states.size() << endl;
16          cout << "Anzahl gefundene einzelne Graphen: " << CO.graphs << " *
                10^" << CO.graphs_pow <<endl;
17          auto stop = chrono::high_resolution_clock::now();
18          auto duration = chrono::duration_cast<chrono::seconds>(stop - start
                );
19          cout << "Execution time: " << duration.count() << "s\n\n";
20
21          //output operator as .dot file for graph visualization
22          fileproc::output_completedOperator(CO, predefEdges, predefStates,
                wnames);
23
24          cin.get();//leave console open
25          return 0;
26      }
```

Listing 16: Namespace modelcheck::

```
namespace model_check {

  bool checkSubTree(const SyntaxTree::node* root, const vector<bool>&
      alpha, const vector<int>& TPO1, const vector<int>& TPO2,
      unordered_map<string, int>& var_assg);

  bool c_OccurCheck(const SyntaxTree::node* root);

  vector<string> getFreeVariables(const SyntaxTree::node* root);

  bool checkAllVarAssigns(const vector<SyntaxTree>& FOBC_Trees, const
      vector<bool>& alpha, const vector<int>& TPO1, const vector<int>&
      TPO2, const int& c_world);

  unordered_map<string, int> genVarAssigns(const int& worlds, const
      vector<string>& freeVars, const int& index);
}
```

Listing 17: Rekursive Suche nach Konstante c

```
bool model_check::c_OccurCheck(const SyntaxTree::node* root)
//recursive check if the constant c is part of a given syntax tree
{
  if (root == nullptr)
    return false;

  if ( find(root->args.begin(), root->args.end(), "c") != root->args.end
      ())
    return true;
  else
    return(c_OccurCheck(root->left) || c_OccurCheck(root->right));
}
```

Listing 18: Rekursiver Erfüllungscheck

```
bool model_check::checkSubTree(const SyntaxTree::node* root, const vector
    <bool>& alpha, const vector<int>& TPO1, const vector<int>& TPO2,
    unordered_map<string, int>& var_assg)
//evaluate if a given formula as a syntax tree is fulfilled by given
    alpha, TPO of origin state, TPO of destination state and variable
    assignment
{
  if (root == nullptr)
    return true;

  if (root->type == "Mod") {
    if (alpha[var_assg[root->args[0]]])
      return true;
```

```
10    }
11    else if (root->type == "TPO1") {
12      if (TPO1[var_assg[root->args[0]]] <= TPO1[var_assg[root->args[1]]])
13        return true;
14    }
15    else if (root->type == "TPO2") {
16      if (TPO2[var_assg[root->args[0]]] <= TPO2[var_assg[root->args[1]]])
17        return true;
18    }
19    else if (root->type == "~") {
20      return !checkSubTree(root->left, alpha, TPO1, TPO2, var_assg);
21    }
22    else if (root->type == "E") {
23      for (auto x = 0; x < TPO1.size(); x++) {
24        var_assg[root->args[0]] = x;
25        if (checkSubTree(root->left, alpha, TPO1, TPO2, var_assg))
26          return true;
27      }
28    }
29    else if (root->type == "A") {
30      for (auto x = 0; x < TPO1.size(); x++) {
31        var_assg[root->args[0]] = x;
32        if (!checkSubTree(root->left, alpha, TPO1, TPO2, var_assg))
33          return false;
34      }
35
36      return true;
37    }
38    else{
39      bool left = checkSubTree(root->left, alpha, TPO1, TPO2, var_assg);
40      bool right = checkSubTree(root->right, alpha, TPO1, TPO2, var_assg);
41
42      if (root->type == "&")
43        return (left && right);
44
45      else if (root->type == "|")
46        return (left || right);
47
48      else if (root->type == "=>")
49        return (!left || right);
50
51      else if (root->type == "<=>")
52        return (left == right);
53    }
54
55    return false;
56  }
```

Literaturverzeichnis

[1] C. Beierle and G. Kern-Isberner, *Methoden wissensbasierter Systeme*, 2019, 6th edition.

[2] B. Renz, "Logik und formale methoden. vorlesungsskript wintersemester 2019/2020," 2019. [Online]. Available: http://eudml.org/doc/168546

[3] T. Aravanis, P. Peppas, and M.-A. Williams, "Observations on darwiche and pearl's approach for iterated belief revision," 08 2019, pp. 1509–1515.

[4] S. O. Hansson, "Logic of belief revision," in *The Stanford Encyclopedia of Philosophy*, winter 2017 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2017.

[5] S. Rosenkranz, *Elementare Prädikatenlogik*. Stuttgart: J.B. Metzler, 2006, pp. 102–173. [Online]. Available: https://doi.org/10.1007/978-3-476-05048-9_3

[6] A. Oberschelp, *Prädikatenlogik*. Stuttgart: J.B. Metzler, 1997, pp. 67–178. [Online]. Available: https://doi.org/10.1007/978-3-476-03628-5_3

[7] D. Jacquette, *Philosophy, Psychology, and Psychologism: Critical and Historical Readings on the Psychological Turn in Philosophy*. Dordrecht: Springer Netherlands, 2003. [Online]. Available: https://doi.org/10.1007/0-306-48134-0

[8] R. van Riel and G. Vosgerau, *Grundlagen der Prädikatenlogik*. Stuttgart: J.B. Metzler, 2018, pp. 103–116. [Online]. Available: https://doi.org/10.1007/978-3-476-04565-2_9

[9] S. Nienhuys-Cheng, S. De Wolf, R. de Wolf, R. Wolf, J. Carbonell, and J. Siekmann, *First-order logic*, ser. Lecture Notes in Artificial Intelligence. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 17–34. [Online]. Available: https://doi.org/10.1007/3-540-62927-0_2

[10] P. A. Flach, *First-Order Logic*. Boston, MA: Springer US, 2010, pp. 410–415. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_311

[11] D. Kumar and K. K. Mishra, "Incorporating logic in artificial bee colony (abc) algorithm to solve first order logic problems: The logical abc," in *2015 7th International Conference on Knowledge and Smart Technology (KST)*, Jan 2015, pp. 65–70.

[12] G. Boole, "The calculus of logic," *Studia Philosophica. Commentarii Societatis Philosophicae Polonorum*, vol. 3, no. 1848, pp. 183 – 198, 1848.

[13] G. Frege, *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle: Verlag von Louis Nebert, 1879. [Online]. Available: http://resolver.sub.uni-goettingen.de/purl?PPN538957069, http://gallica.bnf.fr/ark:/12148/bpt6k65658c

[14] A. North Whitehead and B. Russell, *Principia Mathematica*. Cambridge, UK: Cambridge University Press, 1910-1913.

[15] J. A. Robinson, “A machine-oriented logic based on the resolution principle,” *J. ACM*, vol. 12, no. 1, p. 23–41, Jan. 1965. [Online]. Available: https://doi.org/10.1145/321250.321253

[16] D. Hilbert and R. Courant, *Methods of Mathematical Physics*. CUP Archive, 1966, vol. 1.

[17] H. C. Kennedy, *Life and works of Giuseppe Peano*. Dodrecht: D. Reidel Publ. Co, 1980.

[18] K. Gödel, “Über die vollständigkeit des logikkalküls,” Ph.D. dissertation, 1929.

[19] G. Gentzen, “Untersuchungen über das logische schließen i,” *Mathematische Zeitschrift*, vol. 39, pp. 176–210, 1935. [Online]. Available: http://eudml.org/doc/168546

[20] A. Tarski, “Der Wahrheitsbegriff in den formalisierten Sprachen,” *Studia Philosophica. Commentarii Societatis Philosophicae Polonorum*, pp. 261 – 405, 1935. [Online]. Available: https://www.sbc.org.pl/dlibra/publication/24411/edition/21615

[21] ——, “Logic, Semantics, Metamathematics. Papers from 1923 to 1938,” 1956.

[22] A. Church, “An unsolvable problem of elementary number theory,” *American Journal of Mathematics*, vol. 58, no. 2, pp. 345–363, Apr. 1936. [Online]. Available: http://dx.doi.org/10.2307/2371045

[23] B. A. Smith and R. W. Wilkerson, “Fault diagnosis using first order logic tools,” in *Proceedings of the 32nd Midwest Symposium on Circuits and Systems,*, Aug 1989, pp. 299–302 vol.1.

[24] C. Peirce, C. Hartshorne, P. Weiss, and A. Burks, *Collected Papers of Charles Sanders Peirce*, ser. Collected Papers of Charles Sanders Peirce. Harvard University Press, 1931, no. Bd. 2.

[25] A. L. Reyes-Cabello, A. Aliseda-Llera, and n. Nepomuceno-Fernández, “Towards Abductive Reasoning in First-order Logic,” *Logic Journal of the IGPL*, vol. 14, no. 2, pp. 287–304, 03 2006. [Online]. Available: https://doi.org/10.1093/jigpal/jzk019

[26] F. Baader, B. Beckert, and T. Nipkow, “Deduktion: von der theorie zur anwendung,” *Informatik Spektrum*, vol. 33, pp. 444–451, 10 2010.

[27] J. Fisseler, “First-order probabilistic conditional logic and maximum entropy,” *Logic Journal of the IGPL*, vol. 20, no. 5, pp. 796–830, 03 2012. [Online]. Available: https://doi.org/10.1093/jigpal/jzs008

[28] W. Deng, J. Lai, Y. Xu, X. He, and J. Zhang, “Reasoning under uncertainty based on linguistic truth-valued lattice values first-order logic (ii),” in *2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*, Nov 2009, pp. 1666–1669.

[29] L. Xiao, J. Meng, S. Ding, and L. Zou, “A resolution method for linguistic truth-valued intuitionistic fuzzy first-order logic,” in *2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, July 2016, pp. 904–906.

[30] Wenjiang Li and Shuwei Chen, "An automated reasoning method on first-order tense logic," in *2013 International Conference on Machine Learning and Cybernetics*, vol. 04, July 2013, pp. 1706–1711.

[31] B. Heinemann, "An application of monodic first-order temporal logic to reasoning about knowledge," in *10th International Symposium on Temporal Representation and Reasoning, 2003 and Fourth International Conference on Temporal Logic. Proceedings.*, July 2003, pp. 10–16.

[32] I. Hodkinson, F. Wolter, and M. Zakharyaschev, "Decidable and undecidable fragments of first-order branching temporal logics," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, July 2002, pp. 393–402.

[33] F. Dau, *8 First Order Logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 83–91. [Online]. Available: https://doi.org/10.1007/978-3-540-40062-2_8

[34] J. R. Moschovakis, "Dieter rödding. anzahlquantoren in der prädikatenlogik. archiv für mathematische logik und grundlagenforschung, vol. 9 no. 3–4 (1966), pp. 66–69." *Journal of Symbolic Logic*, vol. 33, no. 3, p. 473–473, 1968.

[35] J. Corcoran and J. Herring, "Heinz-dieter ebbinghaus. Über eine prädikatenlogik mit partiell definierten prädikaten und funktionen. archie für mathematische logik und grundlagenforschung, vol. 12 (1969), pp. 39–53." *Journal of Symbolic Logic*, vol. 37, no. 3, p. 617–618, 1972.

[36] M. J. Gabbay and A. Mathijssen, "One-and-a-halfth-order Logic," *Journal of Logic and Computation*, vol. 18, no. 4, pp. 521–562, 11 2007. [Online]. Available: https://doi.org/10.1093/logcom/exm064

[37] S. SHAPIRO, "Do Not Claim Too Much: Second-order Logic and First-order Logic," *Philosophia Mathematica*, vol. 7, no. 1, pp. 42–64, 02 1999. [Online]. Available: https://doi.org/10.1093/philmat/7.1.42

[38] A. Darwiche and J. Pearl, "On the logic of iterated belief revision," *Artificial Intelligence*, vol. 89, no. 1, pp. 1 – 29, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370296000380

[39] R. Reiter, "A logic for default reasoning," *Artificial Intelligence*, vol. 13, no. 1, pp. 81 – 132, 1980, special Issue on Non-Monotonic Logic. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370280900144

[40] H. Geffner, "Default reasoning - causal and conditional theories," 1992.

[41] M. Goldszmidt and J. Pearl, "Qualitative probabilities for default reasoning, belief revision, and causal modeling," *Artificial Intelligence*, vol. 84, no. 1, pp. 57 – 112, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370295000909

[42] J. Müsseler and M. Rieger, Eds., *Allgemeine Psychologie; 3. Auflage*, ser. Lehrbuch. Berlin: Springer, 2017. [Online]. Available: https://publications.rwth-aachen.de/record/667625

[43] H. Katsuno and A. O. Mendelzon, "Propositional knowledge base revision and minimal change," *Artif. Intell.*, vol. 52, no. 3, pp. 263–294, 1992. [Online]. Available: https://doi.org/10.1016/0004-3702(91)90069-V

[44] C. Alchourrón, P. Gärdenfors, and D. Makinson, "On the logic of theory change: Partial meet contraction and revision functions," *J. Symb. Log.*, vol. 50, pp. 510–530, 06 1985.

[45] M. Grajner and G. Melchior, *Handbuch Erkenntnistheorie*. J.B. Metzler, 2019. [Online]. Available: https://books.google.de/books?id=mFOMDwAAQBAJ

[46] T. Caridroit, S. Konieczny, and P. Marquis, "Contraction in propositional logic," *International Journal of Approximate Reasoning*, vol. 80, pp. 428 – 442, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0888613X16300950

[47] E. Fermé and S. O. Hansson, "Agm 25 years: Twenty-five years of research in belief change," *Journal of Philosophical Logic*, vol. 40, pp. 295–331, 04 2011.

[48] W. Spohn, "Ordinal conditional functions : a dynamic theory of epistemic states," in *Causation in decision, belief change, and statistics*, ser. Proceedings of the Irvine Conference on Probability and Causation, H. William L., Ed., no. 2. Dordrecht: Kluwer, 1988, pp. 105–134.

[49] ——, "A general non-probabilistic theory of inductive reasoning," *Machine intelligence and pattern recognition*, vol. 9, no. Uncertainty in Artificial Intelligence 4, pp. 149–158, 1990.

[50] I. J. Good, "The number of orderings of n candidates when ties are permitted," *Fibonacci Quarterly*, vol. 13, pp. 11–18, 1975. [Online]. Available: https://www.fq.math.ca/Scanned/13-1/good.pdf

Anhang

Source Code

CompleteChangeOperator.cpp

```cpp
#include "input_output_proc.h"
#include <chrono>

int main()
{
  auto start = chrono::high_resolution_clock::now();
  srand(time(NULL));

  vector<string> FOBC_input = fileproc::read_FOBC_input(); // read FOBC input from .txt file
  vector<SyntaxTree> FOBC_trees(FOBC_input.size());;

  for (auto i = 0; i < FOBC_trees.size(); i++) {
    switch (FOBC_trees[i].parse_FOBC(FOBC_input[i])) { // print syntax tree or error (debugging)
      case 0:
        cout << "SyntaxTree " << i+1 << "/" << FOBC_trees.size() << ":\n";
        SyntaxTree::createExpressionString(FOBC_trees[i].root, 0);
        cout << endl << endl;
        break;
      case 1:
        cout << "The following FOBC input has been ignored due to invalid syntax:\n" << FOBC_input[i] << endl << endl;
    }
  }

  vector<string> wnames;
  vector<fileproc::rawState> predefStates = fileproc::get_predefStates(wnames); //get predefined states from first .csv file
  vector<fileproc::rawEdge> predefEdges = fileproc::get_predefEdges(wnames, predefStates); //get predefined edges from second .csv file
  if (wnames.size() < 2 || wnames.size() > 6) { //check if number of worlds is in accepted range
    cout << "Number of worlds out of range.\n";
    cout << "A Minimum of 2 worlds are required to model a believe change.\n";
    cout << "The max amount of worlds is restricted to 6 due to execution time (6 worlds -> >4683 states).\n";
    cin.get();//leave console open
    return 1;
  }

```

```
35    changeOperator CO(wnames.size()); //create changeOperator
36    for (fileproc::rawState rS : predefStates) { CO.addState(rS.TPO); } //
          add predefined states
37    for (fileproc::rawEdge rE : predefEdges) { CO.addEdge(rE.origin, rE.
          edge.dest, rE.edge.alpha); } //add predefined states
38
39
40    switch (CO.complete(FOBC_trees)) { //complete operator and check result
41
42      case 1: //failure
43        cout << "\n\n-------------------------------COMPLETION FAILED
              -------------------------------\n\n";
44        cout << "There exists no complete operator for your input.\n";
45        cin.get();//leave console open
46        return 1;
47
48      case 0: //success
49
50        //CO.debugOutput(); //visualize completed operator in system
              console
51
52        //output summary in system console
53        cout << "\n\n-------------------------SUCESSFULLY FINISHED
              -------------------------\n\n";
54        cout << "Total States: " << CO.states.size() << endl;
55        cout << "Anzahl gefundene einzelne Graphen: " << CO.graphs << " *
              10^" << CO.graphs_pow <<endl;
56        auto stop = chrono::high_resolution_clock::now();
57        auto duration = chrono::duration_cast<chrono::seconds>(stop - start
              );
58        cout << "Execution time: " << duration.count() << "s\n\n";
59
60        //output operator as .dot file for graph visualization
61        fileproc::output_completedOperator(CO, predefEdges, predefStates,
              wnames);
62
63        cin.get();//leave console open
64        return 0;
65    }
66
67  }
```

input output proc.h

```cpp
#pragma once

#include <iostream>
#include <fstream>
#include "belief_change_space.h"

#ifdef __unix
#define fopen_s(pFile,filename,mode) ((*(pFile))=fopen((filename),(mode))) == NULL
#endif

namespace fileproc {

  //--------------------input--------------------//

  struct rawEdge {
    changeOperator::state::edge edge;
    int origin = 0;
  };

  struct rawState {
    vector<int> TPO;
    string name;
  };

  vector<string> read_FOBC_input(void);

  vector<rawState> get_predefStates(vector<string>& wnames);

  vector<rawEdge> get_predefEdges(const vector<string>& wnames, const vector<fileproc::rawState>& states);

  bool is_number(const vector<string>& s);

  bool is_bool(const vector<string>& s);

  bool state_exists(const string& token, const vector<fileproc::rawState>& states);

  bool emptyRow(const string& row);

  bool doubleName(const string& name, const vector<fileproc::rawState>& temp_states);

  int findIndex(const string& s, const vector<fileproc::rawState>& temp_states);

```

```
43    vector<int> normalizeTPO(const vector<int>& input);
44
45    //----------------------output-------------------------------//
46
47    int output_completedOperator(const changeOperator& CO, const vector<
          fileproc::rawEdge>& pdEs, const vector<fileproc::rawState>& pdSs,
          const vector<string>& wnames);
48
49    bool israwEdge(const changeOperator::state::edge& e, const int& i,
          const vector<fileproc::rawEdge>& pdEs);
50
51    string genLabel(const vector<bool>& alpha, const vector<string>& wnames
          );
52
53    string genTPOLabel(const vector<int>& TPO, const vector<string>& wnames
          );
54
55    string getState(const int& index, const vector<fileproc::rawState>&
          predefStates);
56
57 }
```

input output proc.cpp

```
1  #include "input_output_proc.h"
2
3  //----------------------input-------------------------------//
4
5  vector<string> fileproc::read_FOBC_input(void)
6  //read FOBC formualas from input file
7  {
8    ifstream if_FOBC("INPUT_FOBC.txt");
9    vector<string> FOBC_in;
10
11   if (if_FOBC.is_open())
12   {
13     int n = 0;
14     string tempString;
15     while (!if_FOBC.eof())
16     {
17       getline(if_FOBC, tempString);
18       tempString.erase(std::remove_if(tempString.begin(), tempString.end
             (), ::isspace), tempString.end());
19       if (!tempString.empty() && tempString[0] != '*') //skip comments
             and empty lines
20       {
21         // check if nr. of opening and closing brackets match
22         if (!(count(tempString.begin(), tempString.end(), '(') == count(
```

```
            tempString.begin(), tempString.end(), ')')))
23          {
24            cout << "Check brackets. FOBC sentence in line " << (n + 1) <<
                " was ignored.\n\n";
25          }
26          else
27          {
28            FOBC_in.push_back(tempString);
29          }
30        }
31        n++;
32      }
33
34      if_FOBC.close();
35    }
36    else { cout << "FOBC input file failed to load\n\n"; }
37
38    return(FOBC_in);
39  }
40
41  vector<fileproc::rawState> fileproc::get_predefStates(vector<string>&
      wnames)
42  //read predefined states from input file
43  {
44    ifstream if_states("INPUT_States.csv");
45    vector<fileproc::rawState> states_in;
46
47    if (if_states.is_open())
48    {
49      auto n = 0;
50      string tempString;
51      bool headline = true;
52      while (!if_states.eof())
53      {
54        getline(if_states, tempString);
55        tempString.erase(std::remove_if(tempString.begin(), tempString.end
            (), ::isspace), tempString.end());
56        if (!tempString.empty() && tempString[0] != '*' && tempString[0] !=
             ';') //skip comments and empty lines
57        {
58          //remove spaces
59          tempString.erase(std::remove_if(tempString.begin(), tempString.
              end(), ::isspace), tempString.end());
60
61          string delimiter = ";";
62          size_t pos = 0;
63          vector<string> tokens;
64          while ((pos = tempString.find(delimiter)) != string::npos) {
```

```
65            tokens.push_back(tempString.substr(0, pos));
66            tempString.erase(0, pos + delimiter.length());
67          }
68          tokens.push_back(tempString);
69          vector<string> TPOtokens = tokens;
70          TPOtokens.erase(TPOtokens.begin(), TPOtokens.begin() + 1); //
                ignore first column
71
72          if (headline) //get #nr and names of worlds in first line with
                content
73          {
74            wnames = TPOtokens;
75            headline = false;
76          }
77          else
78          {
79            //check if line vector is fully filled with numbers
80            if (TPOtokens.size() != wnames.size() || !fileproc::is_number(
                  TPOtokens)) {
81              cout << "****WARNING!****\nFaulty TPO predefined. Your
                      predefined states (and therefore also edges) were ignored
                      .\n";
82              cout << "TPOs have to be fully defined! Check line " << (n +
                      1) << ".\n\n";
83              return vector<rawState>();
84            }
85            else
86            {
87              if (!doubleName(tokens[0], states_in)) //check if state name
                      is used twice
88              {
89                fileproc::rawState tempstate;
90                tempstate.name = tokens[0];
91
92                vector<int> tempTPO;
93                for (auto i = 0; i < TPOtokens.size(); i++)
94                  tempTPO.push_back(stoi(TPOtokens[i]));
95                tempstate.TPO = fileproc::normalizeTPO(tempTPO);
96
97                states_in.push_back(tempstate);
98              }
99              else {
100               cout << "****WARNING!****\nState defined twice! Second
                        definition of \"" << tokens[0];
101             cout << "\" in line " << (n + 1) << " has been ignored.\n\n
                        ";
102           }
103         }
```

```
104           }
105         }
106         n++;
107       }
108       if_states.close();
109     }
110     else { cout << "Input file for states definition failed to load\n\n"; }
111     return(states_in);
112 }
113
114 vector<fileproc::rawEdge> fileproc::get_predefEdges(const vector<string>&
        wnames, const vector<fileproc::rawState>& states)
115 //read predefined states from input file
116 {
117     ifstream if_edges("INPUT_Edges.csv");
118     vector<fileproc::rawEdge> edges_in;
119
120     if (if_edges.is_open())
121     {
122       auto n = 0;
123       string tempString;
124       bool headline = true;
125       while (!if_edges.eof())
126       {
127         getline(if_edges, tempString);
128         tempString.erase(std::remove_if(tempString.begin(), tempString.end
              (), ::isspace), tempString.end());
129         if (!tempString.empty() && tempString[0] != '*' && !emptyRow(
              tempString)) //skip comments and empty lines
130         {
131           tempString.erase(std::remove_if(tempString.begin(), tempString.
                end(), ::isspace), tempString.end());
132
133           string delimiter = ";";
134           size_t pos = 0;
135           vector<string> tokens;
136           while ((pos = tempString.find(delimiter)) != string::npos) {
137             tokens.push_back(tempString.substr(0, pos));
138             tempString.erase(0, pos + delimiter.length());
139           }
140           tokens.push_back(tempString);
141           if (tokens.size() < 3) //check for minimum length of
142           {
143             cout << "****WARNING!****\nFaulty line found in edge input file
                    (empty data field)! Check line " << (n + 1) << endl;;
144             cout << "Line was ignored.\n\n";
145             n++;
146             continue;
```

```
147         }
148
149         vector<string> data_tokens;
150         data_tokens = tokens;
151         data_tokens.erase(data_tokens.begin(), data_tokens.begin() + 2);
                // erase first 2 columns
152
153         if (headline) {
154           if (wnames != data_tokens) //check if worlds match worlds in
                  states input file
155           {
156             cout << "****WARNING!****\nNames or order of worlds does not
                    match between the two .csv files!\n";
157             cout << "Predefined Edges have been ignored.\n\n";
158             return vector<fileproc::rawEdge>();
159           }
160           headline = false;
161         }
162         else
163         {
164           //check if line is fully filled with boolean values and origin
                  and destination are defined in states input file
165           if (data_tokens.size() != wnames.size() || !is_bool(data_tokens
                  )
166             || !state_exists(tokens[0], states) || !state_exists(tokens
                    [1], states))
167           {
168             cout << "****WARNING!****\nFaulty Edge predefined in line "
                    << (n + 1) << " was ignored.\n";
169             cout << "Edges have to be fully defined and states have to
                    match the definition in the other .csv!\n\n";
170             n++;
171             continue;
172           }
173           else {
174             fileproc::rawEdge tempEdge;
175             tempEdge.origin = findIndex(tokens[0], states);
176             tempEdge.edge.dest = findIndex(tokens[1], states);
177             for (auto i = 0; i < data_tokens.size(); i++) { tempEdge.edge
                    .alpha.push_back(data_tokens[i] == "true"); }
178             edges_in.push_back(tempEdge);
179           }
180         }
181       }
182       n++;
183     }
184     if_edges.close();
185   }
```

```
186   else { cout << "Input file for edges definition failed to load\n\n"; }
187   return(edges_in);
188 }
189
190 bool fileproc::is_number(const vector<string>& s)
191 //check if all elements of string vector consist of only numbers
192 {
193   for (auto i = 0; i < s.size(); i++)
194   {
195     if (s[i].empty() || std::find_if(s[i].begin(),
196       s[i].end(), [](unsigned char c) { return !std::isdigit(c); }) != s[
              i].end())
197     {
198       return false;
199     }
200   }
201
202   return true;
203 }
204
205 bool fileproc::is_bool(const vector<string>& s)
206 //check if all elements of string vector consist of only "true" or "false
      "
207 {
208   bool mintrue = false;
209   for (auto x = 0; x < s.size(); x++)
210   {
211     if (s[x] != "true" && s[x] != "false")
212     {
213       return false;
214     }
215     if (s[x] == "true")
216       mintrue = true;
217   }
218
219   return mintrue;
220
221 }
222
223 bool fileproc::state_exists(const string& token, const vector<fileproc::
      rawState>& states)
224 // check if input string is part of the name attributes in rawState
      vector
225 {
226   for (fileproc::rawState st : states)
227     if (token == st.name)
228       return true;
229
```

```
230   return false;
231 }
232
233 bool fileproc::emptyRow(const string& row)
234 //check if string only consists of semicolons
235 {
236   for (char c : row)
237   {
238     if (c != ';')
239       return false;
240   }
241   return true;
242 }
243
244 bool fileproc::doubleName(const string& name, const vector<fileproc::
      rawState>& temp_states)
245 //check if input string is already part of a name attribute in the
      rawState vector
246 {
247   for (fileproc::rawState s : temp_states)
248     if (s.name == name)
249       return true;
250   return false;
251 }
252
253 int fileproc::findIndex(const string& s, const vector<fileproc::rawState
     >& temp_states)
254 //find the index of the element in the rawState vector, with a name
      attribute that matches the input string
255 {
256   for (auto i = 0; i < temp_states.size(); i++)
257   {
258     if (temp_states[i].name == s)
259       return i;
260   }
261 }
262
263 vector<int> fileproc::normalizeTPO(const vector<int>& input)
264 //normalize input TPO
265 {
266   int max_order = input.size() - 1;
267   vector<int> output = input;
268   for (auto i = 0; i < max_order; i++) {
269     if (count(output.begin(), output.end(), i) == 0) {
270       for (auto n = 0; n < output.size(); n++) {
271         if (output[n] > i)
272           output[n] = output[n] - 1;
273       }
```

```
274       }
275     }
276     return (output);
277 }
278 
279 //----------------------output-----------------------------//
280 
281 int fileproc::output_completedOperator(const changeOperator& CO, const
        vector<fileproc::rawEdge>& pdEs, const vector<fileproc::rawState>&
        pdSs, const vector<string>& wnames)
282 //output a changeOperator object as a .dot graph visualization file with
        different coloring for predefined states and edges specified in the
        three input vectors
283 {
284   FILE* fout;
285   fopen_s(&fout, "completedOperator.dot", "w");
286 
287   //--------------------------------predefined states/edges
          ---------------------------------------//
288   fprintf(fout, "digraph CompletedOperator{\n\nrankdir=LR; concentrate=
        false;\n\n");
289   fprintf(fout, "subgraph Predef {\nnode[color=blue];\nedge[color=blue];\
        n");
290   for (fileproc::rawEdge pdE : pdEs)
291   {
292     fprintf(fout, "\"%s\" -> \"%s\" [label=<<font color=\"blue\">%s</font
          >>]\n", getState(pdE.origin, pdSs).c_str(), getState(pdE.edge.dest
          , pdSs).c_str(), genLabel(pdE.edge.alpha, wnames).c_str());
293   }
294 
295   fprintf(fout, "node[color=blue shape=box style=dashed];\nedge[color=
        blue style=dashed arrowhead=none];\n");
296   for (fileproc::rawState pdS : pdSs)
297   {
298     fprintf(fout, "\"%s\" -> \"%s\"\n", genTPOLabel(pdS.TPO, wnames).
          c_str(), pdS.name.c_str());
299   }
300   fprintf(fout, "}\n\n");
301 
302   //--------------------------------inserted states/edges
          ---------------------------------------//
303   fprintf(fout, "node[style=filled];\nedge[color=grey];\n");
304   for (auto i = 0; i < CO.states.size(); i++) {
305     for (changeOperator::state::edge x : CO.states[i].edges)
306     {
307       if (!israwEdge(x, i, pdEs))
308         fprintf(fout, "\"%s\" -> \"%s\" [label=\"%s\"]\n", getState(i,
            pdSs).c_str(), getState(x.dest, pdSs).c_str(), genLabel(x.
```

```
            alpha, wnames).c_str());
309       }
310     }
311     fprintf(fout, "node[color=black shape=box style=dashed];\nedge[color=
          black style=dashed arrowhead=none];\n");
312     for (auto i = pdSs.size(); i < CO.states.size(); i++)
313       fprintf(fout, "\"%s\" -> \"%s\"\n", genTPOLabel(CO.states[i].TPO,
            wnames).c_str(), getState(i, pdSs).c_str());
314
315     fprintf(fout, "}\n");
316     fclose(fout);
317
318     return EXIT_SUCCESS;
319 }
320
321 bool fileproc::israwEdge(const changeOperator::state::edge& e,const int&
      i, const vector<fileproc::rawEdge>& pdEs)
322 {
323     for (fileproc::rawEdge rE : pdEs)
324     {
325       if (rE.origin == i && rE.edge.alpha == e.alpha)
326         return true;
327     }
328     return false;
329 }
330
331 string fileproc::genLabel(const vector<bool>& alpha, const vector<string
      >& wnames)
332 //convert internal alpha representation (vector of bools) to plain text
      for improved human readability
333 {
334     string output ="{";
335     for (auto i = 0; i < alpha.size(); i++)
336     {
337       if (alpha[i])
338       {
339         if (output != "{")
340           output += ", ";
341         output += wnames[i];
342       }
343     }
344     output += "}";
345     return output;
346 }
347
348 string fileproc::genTPOLabel(const vector<int>& TPO, const vector<string
      >& wnames)
349 //convert internal TPO representation (vector of ints) to plain text for
```

```
    improved human readability
{
  string output = "";
  int maxElement = *std::max_element(TPO.begin(), TPO.end());
  for (auto i = 0; i <= maxElement; i++)
  {
    vector<bool> boolTPO;
    for (auto x = 0; x < TPO.size(); x++) { boolTPO.push_back(TPO[x] == i
        ); }
    output += fileproc::genLabel(boolTPO, wnames);
    if (i != maxElement)
      output += " < ";
  }
  return output;
}

string fileproc::getState(const int& index, const vector<fileproc::
    rawState>& predefStates)
//get predefined state name if existent, otherwise generate a state name
{
  string sname;
  if (index < predefStates.size())
  {
    sname = predefStates[index].name;
  }
  else
  {
    sname = "s";
    sname += to_string(index - predefStates.size());
    sname += "*";
  }
  return sname;
}
```

FOBC parser.h

```
#pragma once

#include <vector>
#include <stack>
#include <string>
#include <algorithm>

#include <iostream> //debugging

using namespace std;

class SyntaxTree {
```

```
13
14  public:
15
16    struct node {
17      node* parent, * left, * right;
18      string type;
19      vector<string> args;
20
21      node() : parent(nullptr), left(nullptr), right(nullptr), type(""),
            args(2, "") {}
22    };
23
24    node* root;
25
26    SyntaxTree() {
27      this->root = nullptr;
28    }
29
30    ~SyntaxTree() {
31      deleteBranch(this->root);
32    }
33
34    int parse_FOBC(const string &s);
35
36    static bool isQuantor(const string& s);
37
38    //output syntax tree to system console (debugging)
39    static void createExpressionString(const node* root, const int& n) {
40      if (root == nullptr)
41        return;
42
43      //cout << "Ebene " << n;
44      for (auto i = 0; i < n; i++)
45        cout << "\t";
46
47      if (root->type == "~" || isQuantor(root->type)) {
48        cout << root->type << " with Argument " <<root->args[0] << endl;
49        createExpressionString(root->left, (n + 1));
50      }
51      else if (!isOperator(root->type)) {
52        cout << root->type << " with Arguments " << root->args[0] << "
              and " << root->args[1] << endl;
53      }
54      else {
55        cout << "L-Branch of " << root->type << endl;
56        createExpressionString(root->left, (n + 1));
57        //cout << "Ebene " << n;
58        for (auto i = 0; i < n; i++)
```

```
59            cout << "\t";
60          cout << "R-Branch of " << root->type << endl;
61          createExpressionString(root->right, (n + 1));
62        }
63        return;
64      }
65
66
67    private:
68
69      static node* createNode(const string& s);
70
71      static bool isOperator(const string& s);
72
73      static bool isValidArg(const string& arg, const int& argcount, const
          string& name);
74
75      static int operatorPrecedence(const string& op);
76
77      static node* createSyntaxTree(const vector<string>& expression);
78
79      static vector<string> tokenize(const string& s);
80
81      static void deleteBranch(node* root);
82
83      static int addToStack(const string& op, stack<node*>& operandStack);
84
85    };
```

FOBC parser.cpp

```
1  #include "FOBC_parser.h"
2
3  int SyntaxTree::parse_FOBC (const string& s)
4  //parse FOBC to SyntaxTree
5  {
6      this->root = createSyntaxTree(tokenize(s));
7      if (this->root == nullptr)
8        { return 1; }
9      return 0;
10 }
11
12 vector<string> SyntaxTree::tokenize(const string& s)
13 //tokenize string to vector of string tokens
14 {
15   vector<string> result;
16
17   const char* exp = s.c_str();
```

```
18    unsigned length = (unsigned)s.length();
19
20    for (auto i = 0; i < length; i++) {
21      if (exp[i] == '(') {
22        result.push_back("(");
23      }
24      else if (exp[i] == ')') {
25        result.push_back(")");
26      }
27      else if (exp[i] == '~') {
28        result.push_back("~");
29      }
30      else if (exp[i] == '&') {
31        result.push_back("&");
32      }
33      else if (exp[i] == '|') {
34        result.push_back("|");
35      }
36      else if (exp[i] == '=' && exp[i + 1] == '>') {
37        result.push_back("=>");
38        i++;
39      }
40      else if (exp[i] == '<' && exp[i + 1] == '=' && exp[i + 2] == '>') {
41        result.push_back("<=>");
42        i = i + 2;
43      }
44      else {
45        auto j = i + 1;
46        while (exp[j] != ')') { j++; };
47        string temp(exp, i, (j - i) + 1);
48        result.push_back(temp);
49        i = j;
50      }
51
52    }
53
54    return result;
55  }
56
57  SyntaxTree::node* SyntaxTree::createSyntaxTree(const vector<string>&
        expression)
58  //create tree of tokenized expression by operator precedences
59  {
60    stack<string> op;
61    stack<node*> result;
62
63    for (auto i = 0; i < expression.size(); i++) {
64      string temp = expression[i];
```

```
65      if (isOperator(temp)) {
66        if (op.empty()) {
67          op.push(temp);
68        }
69        else if (temp == "~" || temp == "(" || isQuantor(temp)) {
70          op.push(temp);
71        }
72        else if (temp == ")") {
73          while (op.top() != "(") {
74            if (addToStack(op.top(), result) == 1)
75              return nullptr;
76            op.pop();
77          }
78          op.pop();
79        }
80        else if (operatorPrecedence(temp) > operatorPrecedence(op.top())) {
81          //collect operators on stack as long as precedence of next
                operator is lower
82          op.push(temp);
83        }
84        else if (operatorPrecedence(temp) <= operatorPrecedence(op.top()))
             {
85          //when operator precedence is lower or equal create subtree from
                last operators with higher precedence
86          if ((op.top() != "~" && !isQuantor(op.top()) && result.size() <
                2)) {
87            return nullptr; //avoid out of range
88          }
89          while (!op.empty() && operatorPrecedence(temp) <=
                operatorPrecedence(op.top())) {
90            if (addToStack(op.top(), result) == 1)
91              return nullptr;
92            op.pop();
93          }
94          op.push(temp);
95        }
96      }
97      else { //predicates are directly added to stack, since no subtrees
           have to be created
98        if (addToStack(temp, result) == 1)
99          return nullptr;
100     }
101   }
102
103   while (!op.empty()) {
104     if (addToStack(op.top(), result) == 1)
105       return nullptr;
106     op.pop();
```

```
107   }
108
109   return result.top();
110 }
111
112 int SyntaxTree::addToStack(const string& op, stack<node*>& operandStack)
113 //create subtree for operator, delete node pointers of operands from
      stack and replace with pointer to subtree
114 {
115   node* root = createNode(op); //create node for the operator or operand
116
117   if (root == nullptr) { //delete all nodes already created
118     for (auto i = 0; i < operandStack.size(); i++) {
119       node* temp = operandStack.top();
120       deleteBranch(temp);
121       operandStack.pop();
122     }
123     return 1;
124   }
125
126   if (isOperator(op)) { //if node is an operator process the operator
        based on unary or binary operation
127     if (op == "~" || isQuantor(op)) {
128       node* operand = operandStack.top(); //get node for the operand
129       operand->parent = root;
130       operandStack.pop(); //pop the operand stack
131       root->left = operand; //assign the operand to the left of the root
            operator
132       root->right = nullptr;
133     }
134     else {
135       node* operand2 = operandStack.top(); //get node for second operand
136       operand2->parent = root;
137       operandStack.pop();
138       node* operand1 = operandStack.top(); //get node for first operand
139       operand1->parent = root;
140       operandStack.pop();
141       root->left = operand1; //assign the operand to the left and right
            of the root operator
142       root->right = operand2;
143     }
144   }
145   operandStack.push(root);
146   return 0;
147 }
148
149 SyntaxTree::node* SyntaxTree::createNode(const string& s)
150 //create node from string input and fill attributes according to type
```

```
151 {
152   node* root = new node;
153 
154   if (isOperator(s) && !isQuantor(s)) { //for operators no arguments are
        needed
155     root->type = s;
156   }
157   else { //for quantors and predicates arguments are needed
158     const char* t = s.c_str();
159     int index = 0;
160     while (t[index++] != '(') {};
161     string name = string(t, 0, index - 1);
162     if (name == "Mod" || name == "TPO1" || name == "TPO2" || isQuantor(
          name))
163       root->type = name;
164     else {
165       delete root;
166       return nullptr;
167     }
168 
169     vector<string> arguments;
170     int argcount = 0;
171     for (auto i = index; i < s.length(); i++) {
172       if (t[i] == ',' || t[i] == ')') {
173         string temp(t, index, i - index);
174         index = i + 1;
175         if (isValidArg(temp, argcount, name)) {
176           root->args[argcount] = temp;
177           argcount++;
178         }
179         else {
180           delete root;
181           return nullptr;
182         }
183       }
184     }
185   }
186   return root;
187 }
188 
189 void SyntaxTree::deleteBranch(node* root)
190 //recursive deletion of each node in subtree
191 {
192   if (root == nullptr)
193     return;
194 
195   deleteBranch(root->left);
196   deleteBranch(root->right);
```

```
197   delete root;
198 }
199
200 bool SyntaxTree::isOperator(const string& s)
201 //check if input string represents an FOL operator
202 {
203   return (s == "(" || s == ")" || s == "~" ||
204     s == "&" || s == "|" || s == "=>" ||
205     s == "<=>" || isQuantor(s));
206 }
207
208 bool SyntaxTree::isQuantor(const string& s)
209 //check if input string represents an FOL quantor
210 {
211   return (s[0] == 'A' || s[0] == 'E');
212 }
213
214 bool SyntaxTree::isValidArg(const string& arg, const int& argcount, const
      string& name)
215 //check if input string arg as argument nr argcount is valid for node
     with type name
216 {
217   if (arg[0] == 'c' || arg[0] == 'u' || arg[0] == 'v' || arg[0] == 'w'
218     || arg[0] == 'x' || arg[0] == 'y' || arg[0] == 'z') { //check for
         allowed variables and constants
219     bool numeric_args = true;
220     for (auto i = 1; i < arg.length(); i++) { //numbers after letters are
          allowed to increase possible variables to use
221       if (!isdigit(arg[i])) {
222         numeric_args = false;
223       }
224     }
225     if (numeric_args && arg.length() <= 3) { //only two numbers are
         allowed to index variables
226
227       if ((name == "Mod" || isQuantor(name)) && argcount < 1) //predicate
           "Mod" and quantors have only one argument
228         return true;
229
230       if ((name == "TPO1" || name == "TPO2") && argcount < 2) //
          predicates "TPO1" and "TPO2" have two arguments
231         return true;
232     }
233   }
234   return false;
235 }
236
237 int SyntaxTree::operatorPrecedence(const string& op)
```

```
238 //get operator precedences as rank integer from name
239 {
240   /*if (op == "(" || op == ")")
241   return 8;*/
242   if (op == "~")
243     return 7;
244   if (op == "&")
245     return 6;
246   if (op == "|")
247     return 5;
248   if (op == "=>")
249     return 4;
250   if (op == "<=>")
251     return 3;
252   if (op == "A")
253     return 2;
254   if (op == "E")
255     return 1;
256   return 0;
257 }
```

belief change space.h

```
1  #pragma once
2
3  #include "FOBC_parser.h"
4  #include "model_checking.h"
5  #include <cmath>
6
7  #define reducedDiskSpace true
8  //when "false", TPOs will be created quicker but stored in a (maybe large
       ) table
9  //when "true", TPOs will take longer to create, but not all orders have
       to be stored
10
11 #define outputMode "all"
12 //when "single", one possible completed operator will be created (always
       starts checking states in numerical order)
13 //when "random", a random possible completed operator will be created
14 //when "all", a completed operator containing all possible edges will be
       created
15
16 class changeOperator {
17
18 public:
19
20   changeOperator(const int& world_count) {
21     this->worlds = world_count;
```

```
22      graphs = 1;
23      graphs_pow = 0;
24    }
25
26    void addState(const vector<int>& TPO);
27
28    void addEdge(const int& orig, const int& dest, const vector<bool>&
          alpha);
29
30    int complete(const vector<SyntaxTree>& FOBC_Trees);
31
32    struct state {
33
34      struct edge {
35        vector<bool> alpha;
36        int dest;
37
38        edge() : alpha(), dest() {}
39        edge(const int& dest_in, const vector<bool>& alpha_in) : alpha(
              alpha_in), dest(dest_in) {}
40      };
41
42      vector<int> TPO;
43      vector<edge> edges;
44
45      state() : TPO(), edges() {}
46      state(const vector<int>& TPO_in) : TPO(TPO_in), edges() {}
47
48    };
49
50    vector <state> states;
51    int graphs;
52    int graphs_pow;
53
54    void debugOutput(void) {
55    //output change operator to system console (debugging)
56      for (auto x = 0; x < this->states.size(); x++)
57      {
58        cout << "TPO State " << x << ": ";
59        vector<int> path = this->states[x].TPO;
60        for (auto i = 0; i < path.size(); i++)
61          cout << path[i] << "  ";
62        cout << endl;
63        cout << "Edges: " << this->states[x].edges.size() << endl;
64
65        for (auto y = 0; y < this->states[x].edges.size(); y++) {
66          cout << "\tEdge " << y << ":" << endl;;
67          cout << "\t\tDestination: " << this->states[x].edges[y].dest <<
```

```
            endl;;
68          cout << "\t\tAlpha: ";
69          vector<bool> path2 = this->states[x].edges[y].alpha;
70          for (auto j = 0; j < path2.size(); j++)
71            cout << path2[j] << "  ";
72          cout << endl;
73        }
74      }
75    }
76
77  private:
78
79    vector<vector<int>> TPOtable;
80    int worlds;
81
82    void genTPOtable(const vector<int>& root_TPO);
83
84    bool genTPOs_v2(int& index, vector<int>& TPO);
85
86    vector<int> getTPO(const int& index);
87
88    bool checkTPO(const vector<int>& TPO);
89
90    vector<bool> generateSubsets(const int& index);
91
92    void completeStates(void);
93
94    int completeEdges(const vector<SyntaxTree>& FOBC_cond);
95
96    int bruteForce(const int& origState, const vector<bool>& alpha, const
          vector<SyntaxTree>& FOBC_cond);
97
98    int poweroften(int x);
99
100 };
```

belief change space.cpp

```cpp
#include "belief_change_space.h"

int changeOperator::complete(const vector<SyntaxTree>& FOBC_Trees)
//complete states and edges of this operator object
{
  completeStates(); //loop 1 for adding missing states
  cout << "\n\noutputMode = " << outputMode << endl;
  cout << "reducedDiskSpace = " << reducedDiskSpace << endl << endl;
  return (completeEdges(FOBC_Trees));  //loop 2 for adding missing edges
}

/***************State/TPO Functions************************/

void changeOperator::completeStates(void)
//complete states of this operator object
{
  int predef_states = this->states.size();
  vector<vector<int>> predef_TPOs;
  for (auto k = 0; k < predef_states; k++) {
    predef_TPOs.push_back(this->states[k].TPO);
  }

  if (reducedDiskSpace) { //generate TPOs memory (table) or cpu efficient
    for (auto i = 0; i < pow(this->worlds, this->worlds); i++)
    {
      vector<int> temp_order(this->worlds, 0);
      if (genTPOs_v2(i, temp_order) && count(predef_TPOs.begin(),
          predef_TPOs.end(), temp_order) == 0)
      //for all possible orders, that are not predefined, create a state
      {
        addState(temp_order);
      }
    }
  }
  else {
    vector<int> initialTPO(this->worlds, 0);
    this->TPOtable.push_back(initialTPO);
    genTPOtable(initialTPO);
    for (auto i = 0; i < this->TPOtable.size(); i++)
    {
      vector<int> temp_order = this->TPOtable[i];
      if (count(predef_TPOs.begin(), predef_TPOs.end(), temp_order) == 0)
      //for all possible orders, that are not predefined, create a state
      {
        addState(temp_order);
      }
    }
```

```
47    }
48  }
49
50  void changeOperator::addState(const vector<int>& TPO)
51  //create new state for this object with the input vector as TPO
52  {
53    state new_state(TPO);
54    this->states.push_back(new_state);
55  }
56
57  void changeOperator::genTPOtable(const vector<int>& root_TPO)
58  //generate table for this object containing all possible TPOs depending
      on the nr. of worlds
59  //tree-like search pattern (cpu efficient, not memory efficient)
60  {
61    int max_order = this->worlds - 1;
62    for (auto i = 0; i < this->worlds; i++) {
63      if ((root_TPO[i] < max_order) && (count(root_TPO.begin(), root_TPO.
          end(), root_TPO[i]) > 1)) {
64        vector<int> mod_TPO = root_TPO;
65        mod_TPO[i]++;
66        if (count(this->TPOtable.begin(), this->TPOtable.end(), mod_TPO) ==
            0) {
67          this->TPOtable.push_back(mod_TPO);
68          genTPOtable(mod_TPO);
69        }
70      }
71    }
72    return;
73  }
74
75  bool changeOperator::genTPOs_v2(int& index, vector<int>& TPO)
76  //generate possible TPO with input index (memory efficient, not cpu
      efficient)
77  {
78    while (index < pow(this->worlds, this->worlds))
79    {
80      TPO = getTPO(index);
81      if (checkTPO(TPO))
82        return true;
83      else
84        index++;
85    }
86    return false;
87
88  }
89
90  vector<int>changeOperator::getTPO(const int& index)
```

```
91 //creates candidates for TPOs (vector of integers between 0 and nr. of
       worlds - 1)
92 {
93   int num = index;
94   vector<int> output(this->worlds, 0);
95   for (auto i = 0; i < this->worlds; i++)
96   {
97     output[i] = num % this->worlds;
98     num = num / this->worlds;
99     if (num == 0)
100      break;
101  }
102
103  return output;
104 }
105
106 bool changeOperator::checkTPO(const vector<int>& TPO)
107 //check if input TPO candidate is a standardized TPO
108 {
109  for (auto i = 0; i < this->worlds - 1; i++) {
110    if (count(TPO.begin(), TPO.end(), i) == 0) {
111      for (auto n = 0; n < TPO.size(); n++) {
112        if (TPO[n] > i)
113          return false;
114      }
115    }
116  }
117
118  return true;
119 }
120
121 /***************Edge/Alpha Functions**********************/
122
123 int changeOperator::completeEdges(const vector<SyntaxTree>& FOBC_cond)
124 //complete edges of this operator object
125 {
126  for (auto i = 0; i < this->states.size(); i++) { //check all origin
       states
127
128    int predef_edges = this->states[i].edges.size();
129    vector<vector<bool>> predef_alphas;
130    for (auto k = 0; k < predef_edges; k++) {
131      predef_alphas.push_back(this->states[i].edges[k].alpha);
132    }
133
134    int maxSubsets = pow(2, this->worlds);
135    for (auto j = 1; j < maxSubsets; j++) {
136      vector<bool> new_alpha = generateSubsets(j);
```

```
137       if (count(predef_alphas.begin(), predef_alphas.end(), new_alpha) ==
              0)
138       {
139         //check all alphas for each origin state that are not predefined
140         if (bruteForce(i, new_alpha, FOBC_cond) == 1)
141           return 1;
142       }
143     }
144   }
145
146   return 0;
147 }
148
149 int changeOperator::bruteForce(const int& origState, const vector<bool>&
      alpha, const vector<SyntaxTree>& FOBC_cond)
150 //try all variations of c and destination state for given alpha and
      origin to fulfill input syntax trees
151 {
152   vector<int> TPO1 = this->states[origState].TPO;
153   int c_assign = 1;
154   int edgesfound = 0;
155   vector<bool> dest_checked(states.size(), false);
156
157   //if c is not part of any input FOBC sentence it does not have to be
        varied
158   for (auto l = 0; l < FOBC_cond.size(); l++) {
159     if (model_check::c_OccurCheck(FOBC_cond[l].root)) {
160       c_assign = this->worlds;
161       break;
162     }
163   }
164
165   for (int m = 0; m < this->states.size(); m++)  //loop 3: try all
        destination states
166   {
167     int dest;
168     if (outputMode == "random")
169       while (true)
170       {
171         int r = rand() % states.size(); //select random destination
172         if (!dest_checked[r]) {
173           dest = r;
174           dest_checked[r] = true;
175           break;
176         }
177       }
178     else
179       dest = m;
```

```
180
181     /*if (dest == origState)
182       continue;*/ //when origin must not be destination state
183
184     vector<int> TPO2 = this->states[dest].TPO;
185     for (auto j = 0; j < c_assign; j++) //loop 4: try all assignements
            for c if it is part of the FOBC sentences
186     {
187       if (model_check::checkAllVarAssigns(FOBC_cond, alpha, TPO1, TPO2, j
            ))
188       //check all variable assignments for given combination of syntax
            trees, alpha, c assignment, TPO of destination and TPO of origin
189       {
190         addEdge(origState, dest, alpha);
191         if (outputMode != "all")
192           return 0;
193         else
194           edgesfound++;
195           break;
196       }
197     }
198   }
199
200   if (edgesfound > 0) {
201     this->graphs *= edgesfound;
202     if (this->graphs >= 100) {
203       this->graphs_pow += (poweroften(this->graphs) - 1);
204       this->graphs /= pow(10, poweroften(this->graphs) - 1);
205     }
206     return 0;
207   }
208   else {
209     this->graphs = 0;
210     this->graphs_pow = 0;
211     return 1;
212   }
213 }
214
215 int changeOperator::poweroften(int x)
216 {
217   return x >= 10 ? poweroften(x / 10) + 1 : 0;
218 }
219
220 void changeOperator::addEdge(const int& orig, const int& dest, const
      vector<bool>& alpha)
221 //add edge to object and fill attributes origin, destination and alpha
      with input
222 {
```

```
223   state::edge new_edge(dest, alpha);
224   this->states[orig].edges.push_back(new_edge);
225   return;
226
227 }
228
229 vector<bool> changeOperator::generateSubsets(const int& index)
230 //creates possible subsets of nr. of worlds of object (vector of bools)
231 {
232   int num = index;
233   vector<bool> output(this->worlds, 0);
234   for (auto i = 0; i < this->worlds; i++) {
235     output[i]= num % 2;
236     num = num / 2;
237     if (num == 0)
238       break;
239   }
240
241   return output;
242 }
```

model checking.h

```
1  #pragma once
2
3  #include "belief_change_space.h"
4  #include <unordered_map>
5
6  namespace model_check {
7
8    bool checkSubTree(const SyntaxTree::node* root, const vector<bool>&
       alpha, const vector<int>& TPO1, const vector<int>& TPO2,
       unordered_map<string, int>& var_assg);
9
10   bool c_OccurCheck(const SyntaxTree::node* root);
11
12   vector<string> getFreeVariables(const SyntaxTree::node* root);
13
14   bool checkAllVarAssigns(const vector<SyntaxTree>& FOBC_Trees, const
       vector<bool>& alpha, const vector<int>& TPO1, const vector<int>&
       TPO2, const int& c_world);
15
16   unordered_map<string, int> genVarAssigns(const int& worlds, const
       vector<string>& freeVars, const int& index);
17 }
```

model checking.cpp

```cpp
#include "model_checking.h"


bool model_check::checkSubTree(const SyntaxTree::node* root, const vector<bool>& alpha, const vector<int>& TPO1, const vector<int>& TPO2, unordered_map<string, int>& var_assg)
//evaluate if a given formula as a syntax tree is fulfilled by given alpha, TPO of origin state, TPO of destination state and variable assignment
{
  if (root == nullptr)
    return true;

  if (root->type == "Mod") {
    if (alpha[var_assg[root->args[0]]])
      return true;
  }
  else if (root->type == "TPO1") {
    if (TPO1[var_assg[root->args[0]]] <= TPO1[var_assg[root->args[1]]])
      return true;
  }
  else if (root->type == "TPO2") {
    if (TPO2[var_assg[root->args[0]]] <= TPO2[var_assg[root->args[1]]])
      return true;
  }
  else if (root->type == "~") {
    return !checkSubTree(root->left, alpha, TPO1, TPO2, var_assg);
  }
  else if (root->type == "E") {
    for (auto x = 0; x < TPO1.size(); x++) {
      var_assg[root->args[0]] = x;
      if (checkSubTree(root->left, alpha, TPO1, TPO2, var_assg))
        return true;
    }
  }
  else if (root->type == "A") {
    for (auto x = 0; x < TPO1.size(); x++) {
      var_assg[root->args[0]] = x;
      if (!checkSubTree(root->left, alpha, TPO1, TPO2, var_assg))
        return false;
    }

    return true;
  }
  else{
    bool left = checkSubTree(root->left, alpha, TPO1, TPO2, var_assg);
    bool right = checkSubTree(root->right, alpha, TPO1, TPO2, var_assg);
```

```
44
45      if (root->type == "&")
46        return (left && right);
47
48      else if (root->type == "|")
49        return (left || right);
50
51      else if (root->type == "=>")
52        return (!left || right);
53
54      else if (root->type == "<=>")
55        return (left == right);
56    }
57
58    return false;
59  }
60
61  bool model_check::c_OccurCheck(const SyntaxTree::node* root)
62  //recursive check if the constant c is part of a given syntax tree
63  {
64    if (root == nullptr)
65      return false;
66
67    if ( find(root->args.begin(), root->args.end(), "c") != root->args.end
        ())
68      return true;
69    else
70      return(c_OccurCheck(root->left) || c_OccurCheck(root->right));
71  }
72
73  vector<string> model_check::getFreeVariables(const SyntaxTree::node* root
      )
74  //recursively collect all free variables in a given syntax tree as a
      vector of strings of variable names
75  {
76    if (root == nullptr)
77      return vector<string>();
78
79    vector<string> output;
80
81    if (root->type == "Mod") {
82      if (root->args[0] != "c")
83        output.push_back(root->args[0]);
84    }
85    else if (root->type == "TPO1" || root->type == "TPO2") {
86      if (root->args[0] != "c")
87        output.push_back(root->args[0]);
88      if (root->args[1] != "c")
```

```
89        output.push_back(root->args[1]);
90    }
91    else {
92      vector<string> temp_Variables_L = getFreeVariables(root->left);
93      if (SyntaxTree::isQuantor(root->type) || (root->type == "~")) {
94        for (auto i = 0; i < temp_Variables_L.size(); i++) {
95          if (temp_Variables_L[i] != root->args[0])
96            output.push_back(temp_Variables_L[i]);
97        }
98      }
99      else {
100       output = getFreeVariables(root->right);
101       for (auto i = 0; i < temp_Variables_L.size(); i++) {
102         if (count(output.begin(), output.end(), temp_Variables_L[i]) ==
              0)
103           output.push_back(temp_Variables_L[i]);
104       }
105     }
106   }
107
108   return output;
109
110 }
111
112 bool model_check::checkAllVarAssigns(const vector<SyntaxTree>& FOBC_Trees
      , const vector<bool>& alpha, const vector<int>& TPO1, const vector<int
    >& TPO2, const int& c_world)
113 //check all variable assignments for given combination of syntax trees,
      alpha, c assignment, TPO of destination and TPO of origin
114 {
115   for (auto k = 0; k < FOBC_Trees.size(); k++) {
116   //loop 5: check all FOBC sentences
117     vector<string> freeVariables = model_check::getFreeVariables(
        FOBC_Trees[k].root);
118     unordered_map<string, int> var_assignment;
119
120     for (auto index = 0; index < pow(TPO1.size(), freeVariables.size());
        index++) {
121     //loop 6: check all possible variable assignments
122       var_assignment = model_check::genVarAssigns(TPO1.size(),
          freeVariables, index);
123       var_assignment["c"] = c_world;
124       if (!model_check::checkSubTree(FOBC_Trees[k].root, alpha, TPO1,
          TPO2, var_assignment)) {
125         return false;
126       }
127     }
128   }
```

```
129
130   return true;
131 }
132
133 unordered_map<string, int> model_check::genVarAssigns(const int& worlds,
      const vector<string>& freeVars, const int& index)
134 //generate a possible variable assignment with given index for given
      vector of free variables as strings
135 {
136   int num = index;
137   unordered_map<string, int> var_assign;
138   for (auto i = 0; i < freeVars.size(); i++) {
139     var_assign[freeVars[i]] = num % worlds;
140     num = num / worlds;
141   }
142   return var_assign;
143 }
```

README-Datei

```
**Vervollstaendigung von partiell spezifizierten Wissensaenderungsoperatoren**

Diese SW dient der Vervollstaendigung von partiell spezifizierten Wissensaenderungsoperatoren.

---

Input:
Deterministischer endlicher Aenderungsraum C ueber Omega und Formeln psi ueber einer FO-BC-Signatur, so dass C |= psi gilt.

Output:
Ein vollstaendiger deterministischer endlicher Aenderungsraum C ueber Omega, der C komplett enthaelt und fuer den C|= psi gilt.
Fehlermeldung, falls ein solcher Aenderungsraum nicht existiert.

---

How to use:
- Befuellen der Datei "INPUT_States.csv" mit den Zustaenden inkl. Ordnungen des Eingabe-Aenderungsraums
- Befuellen der Datei "INPUT_Edges.csv" mit den Kanten inkl. Alpha des Eingabe-Aenderungsraums
- Befuellen der Datei "INPUT_FOBC.txt" mit den FOBC Bedingungen fuer die Vervollstaendigung
- Setzen der Optionen "reducedDiskSpace" und "outputMode" in der Datei "belief_change_space.h" und kompilieren des Codes
  Alternativ Wahl des entsprechendes Executables
- Ausfuehren des Programms im selben Verzeichnis wie die drei Input Dateien
- Ausgabedatei "completedOperator.dot" zur Visualisierung des vervollstaendigten Aenderungsraums wird erstellt
- Ueberpruefung der erstellten Syntaxbaeume und Fehlermeldungen in der Konsolenausgabe

---

- Datei "INPUT_States.csv": Zeilen, die mit einem Stern "*" beginnen, werden vom Programm als Kommentarzeilen ignoriert.
        Die erste Spalte muss mit den Namen der vordefinierten Zustaende gefuellt werden.
        Die erste Zeile muss mit den Namen der der betrachteten Welten gefuellt werden. Das Programm
        definiert aus dieser Zeile die Anzahl an Welten.
```

```
32          Die so aufgespannte Matrix aus den Namen der Zustaenden und Namen
                der Welten muss vollstaendig
33          mit natuerlichen Zahlen inkl. 0 befuellt werden. Diese
                repraesentieren den Plausibilitaetsindex
34          der jeweiligen Welt, wobei 0 am plausibelsten ist (Modelle).
35          Jede Zeile der Matrix definiert somit einen Zustand mit
                zugehoeriger Ordnung des Aenderungsraums.
36
37  - Datei "INPUT_Edges.csv": Zeilen, die mit einem Stern "*" beginnen,
        werden vom Programm als Kommentarzeilen ignoriert.
38          Jede Zeile steht fuer eine vordefinierte Kante des
                Aenderungsraums.
39          Die erste Spalte muss mit den Namen der Ausgangszustaende der
                Kanten gefuellt werden.
40          Die zweite Spalte muss mit den Namen der Zielzustaende der Kanten
                gefuellt werden.
41          Die angegebenen Namen der betrachteten Welten muss (auch in der
                Reihenfolge) mit der ersten
42          Zeile der "INPUT_States.csv" uebereinstimmen.
43          Die so aufgespannte Matrix aus den Namen der Ausgangs-/
                Zielzustaende und der Namen der Welten
44          muss vollstaendig mit Werten aus jeweils "true" oder "false"
                befuellt werden, je nachdem ob die
45          jeweilige Welt Modell von der Information der Kante ist. Jede
                Zeile muss mindestens ein "true"
46          enthalten, da neue Informationen ohne Modelle nicht akzeptiert
                werden.
47          Es ist darauf zu achten, dass nur Namen fuer Zustaende verwendet
                werden, welche vorher in der
48          "INPUT_States.csv" Datei definiert wurden.
49
50  - Datei "INPUT_FOBC.txt": Zeilen, die mit einem Stern "*" beginnen,
        werden vom Programm als Kommentarzeilen ignoriert.
51          Folgende Operatoren stehen zur Verfuegung fuer die Formeln: &, |,
                ~, =>, <=>, A(x), E(x).
52          Folgende Praedikate stehen zur Verfuegung:
53            - Mod(x) -> WAHR, wenn Argument x Teil von Mod(a) ist
54            - TPO1(x,y) -> WAHR, wenn x kleiner gleich y in Quasiordnung
                  des Ausgangszustandes der Kante
55            - TPO2(x,y) -> WAHR, wenn x kleiner gleich y in Quasiordnung
                  des Ausgangszustandes der Kante
56          Die Konstante "c" kann verwendet werden, sowie Variablen [u, v
                ,..., z]. Die Variablen koennen
57          ausserdem noch mit bis zu zwei Nummern erweitert werden, z.B. u1,
                x15, w79.
58          Folgende Bindungsprioritaeten sind festgelegt: (stark) ~, &, |,
                =>, <=>, A, E (schwach)
```

```
59          Es kann eine beliebige Anzahl an Formeln eingegeben werden.
                Leerzeichen werden vom Algorithmus
60          automatisch entfernt.
61
62  - Option "reducedDiskSpace":  = false --> totale Quasiordnungen zur
        Vervollstaendigung der Zustaende im Aenderungsraum werden
63            CPU-effizient, d.h. schneller generiert und verarbeitet durch
                  einmalige Generierung
64            einer Tabelle durch baumartige Suche. Benoetigt allerdings
                  potentiell viel Speicherplatz
65            bei hoher Anzahl von Welten.
66          = true --> totale Quasiordnungen werden jeweils wenn benoetigt im
                Algorithmus neu generiert.
67            Dadurch entfaellt der evtl. grosse Speicherbedarf fuer eine
                  Tabelle. Dafuer verlangsamt
68            sich die Ausfuehrung.
69
70  - Option "outputMode":    = "single" --> Der Brute-Force Algorithmus
        betreibt die Suche nach einem neuen Zielzustand
71            fuer eine fehlende Kante immer chronologisch und bricht bei der
                  ersten gefundenen
72            Kante ab.
73          = "random" --> Der Brute-Force Algorithmus waehlt zufaellige
                Zustaende als naechstzupruefend aus
74            und bricht ebenfalls bei der ersten gefundenen Kante ab.
75          = "all" --> Der Brute-Force Algorithmus sucht alle moeglichen
                Kanten und fuegt sie dem Aenderungs-
76            raum hinzu. In diesem Modus wird in der Konsolenausgabe die
                  Anzahl an moeglichen
77            vollstaendigen Aenderungsraeumen fuer die Eingabe ausgegeben.
78
79  - Datei "completedOperator.dot": Die erstellte Datei im .dot Format
        zeigt den erstellten Aenderungsraum.
80            Vordefinierte Zustaende und Kanten sind dabei blau gefaerbt und
                  besitzen ihren
81            vordefinierten Namen. Vom Algorithmus hinzugefuegte Zustaende
                  und Kanten sind grau
82            hinterlegt. Zustaende besitzen Namen "s0*, s1*, ....". Die
                  Welten besitzen die vordefinierten
83            Namen aus den Eingabedateien. Die Visualisierung des .dot
                  Formats ist beispielsweise hier
84            online moeglich: </https://dreampuf.github.io/GraphvizOnline/>
85
86  - Konsolenausgabe:    Das Programm faengt einige Fehler bei der Eingabe
        ab. In diesem Fall wird die Ausfuehrung meist
87          trotzdem fortgesetzt und die fehlerhafte Eingabe ignoriert.
                Deshalb ist es wichtig die Fehler-
```

```
88          meldungen zu pruefen. Ausserdem wird eine Repraesentation der
                aufgestellten Syntaxbaeume ausgegeben.
89          Mit Hilfe dieser kann nochmal geprueft werden, ob die FOBC
                Eingaben korrekt geparst wurden.
90
91  ------------------------------------------------------------------------
92
93  (c) Marco Stock, 06.10.20
94    Masterarbeit, Studiengang: Praktische Informatik
95    Fernuniversitaet in Hagen
96    Betreuer: Prof. Dr. Christoph Beierle
```

Selbstständigkeitserklärung

Ich erkläre, dass ich die Masterarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

_____

Marco Stock

Schweinfurt, 04.11.2020