Implementierung eines Entscheidungsverfahrens für das Bewachte Fragment der Prädikatenlogik

Diplomarbeit

Jan Hladik

Lehr- und Forschungsgebiet
Theoretische Informatik
RWTH Aachen
Prof. Dr.-Ing. Franz Baader

Zuerst möchte ich allen Menschen danken, die mich bei der Erstellung dieser Arbeit und im gesamten Studium unterstützt haben. Ohne sie wäre es mir sicherlich nicht gelungen, es in dieser Form abzuschließen.

Uli Sattler und Stephan Tobies haben mich vom Anfang meiner Hiwi-Tätigkeit bis zum Abschluß dieser Arbeit ständig unterstützt und jederzeit die Fragen beantwortet, die ich hatte. Professor Franz Baader hat es mir ermöglicht, diese Arbeit mit allen denkbaren Freiheiten am Lehr- und Forschungsgebiet Theoretische Informatik zu erstellen. Ohne Sebastian Brandt hätte das Layout seinen Namen nicht verdient.

Jens Baalmann, Marco Bormann, Marc Schirski und Peter Santo waren in den unterschiedlichen Abschnitten meines Studiums gute Freunde, mit denen ich gerne gearbeitet und gerne gefeiert habe.

Meine Eltern haben mir das Studium erst ermöglicht und mich zu keiner Zeit unter Druck gesetzt. Und meine Schwester Gitta war immer für mich da, wenn ich Hilfe brauchte.

Danke!

Ich versichere, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 26. September 2001

Inhaltsverzeichnis

Kapitel 1

Einführung

In dieser Diplomarbeit werden die Implementierung und die Leistungsanalyse von SAGA, einem Erfüllbarkeitsalgorithmus für das *Bewachte Fragment* (**S**atisfiability **A**lgorithm for the **G**u**a**rded Fragment) dargestellt.

Das Bewachte Fragment (*Guarded Fragment*, $\mathcal{GF}$) ist ein ausdrucksstarkes, entscheidbares Fragment der Prädikatenlogik erster Stufe (FO), das zu Beginn der 90er Jahre erstmals beschrieben [AvBN98] und seitdem intensiv untersucht wurde (siehe z. B. [vB97, Grä99b, Grä99a, dN98]). Es ist eine Erweiterung der Multimodallogik $\mathbf{K}_n$ [BdRV01]: Während in der Übersetzung von $\mathbf{K}_n$ (Formel 1.1) in FO nur zweistellige Relationen vorkommen (Formel 1.2), gibt es in $\mathcal{GF}$ keine Einschränkung für die Stelligkeit der Prädikate (Formel 1.3).

$$[i]\varphi \qquad \langle i\rangle\varphi \tag{1.1}$$

$$\forall x(R(c,x) \rightarrow \varphi(x)) \qquad \exists x(R(c,x) \wedge \varphi(x)) \tag{1.2}$$

$$\forall x,y(S(c,d,x,y) \rightarrow \varphi(c,d,x,y)) \qquad \exists x,y(S(c,d,x,y) \wedge \varphi(c,d,x,y)) \tag{1.3}$$

$\mathbf{K}_n$ und andere Modallogiken haben viele für das Entscheidungsproblem nützliche Eigenschaften, etwa die Endliche-Modell-Eigenschaft oder die Baum-Modell-Eigenschaft (siehe Abschnitt 3.1). Die Motivation für die Entwicklung von $\mathcal{GF}$ war, zu verstehen, warum die Modallogik diese Eigenschaften hat, und darauf aufbauend die Ausdrucksstärke der Modallogik unter Beibehaltung dieser Eigenschaften zu erweitern.

Den Modallogiken eng verwandt sind die in der Wissensrepräsentation verwendeten *Beschreibungslogiken* (Description Logics, DL; für eine Einführung siehe z. B. [Baa99]): So ist die Beschreibungslogik $\mathcal{ALC}$ eine syntaktische Variante der Multimodallogik $\mathbf{K}_n$ [Sch91]. (Im Umkehrschluß kann man $\mathcal{GF}$ als Erweiterung von $\mathcal{ALC}$ um n-stellige Rollen auffassen.)

Beschreibungslogiken werden in Anwendungen der Wissensrepräsentation vielfach eingesetzt, siehe z. B. [Sat98, DFvH+00]. Dabei sind *Tableau-Algorithmen* [BS01] eine wichtige Klasse von effizienten Schlußfolgerungsverfahren: Sie entscheiden Erfüllbarkeit einer Formel φ, indem sie versuchen, für φ eine endliche Struktur, genannt *Tableau*, zu erzeugen, aus dem sich ein Modell für φ konstruieren läßt.

Für die Optimierung von Tableau-Algorithmen gibt es zahlreiche Mechanismen [Hor97, HST00]. Eine wesentliche Motivation für die Entwicklung von SAGA war die Beantwortung der Frage, ob die Verwandtschaft zwischen Modallogiken und Beschreibungslogiken so weit reicht, daß die Mechanismen, die das Schließen für Beschreibungslogiken effizient machen und damit zu praktischer Entscheidbarkeit führen, sich auch für $\mathcal{GF}$ bewähren.

1.1 Übersicht der folgenden Kapitel

In Kapitel 2 werden kurz die Funktionsweise von Tableau-Algorithmen und ihre Rolle in der Wissensrepräsentation skizziert. Kapitel 3 beschreibt dann ausführlich Syntax und Eigenschaften von $\mathcal{GF}$ und verwandten Logiken. Der dieser Arbeit zugrundeliegende Tableau-Algorithmus für $\mathcal{GF}$ wird in Kapitel 4 in groben Zügen wiedergegeben.

Die folgenden vier Kapitel bilden den eigentlichen Kern dieser Arbeit: In Kapitel 5 wird die Implementierung detailliert beschrieben. Hier ist der Algorithmus noch möglichst einfach gehalten, während in Kapitel 6 die in SAGA eingesetzten, größtenteils von anderen DL-Implementierungen [Hor97, Hla00] übernommenen Optimierungen und Heuristiken dargestellt werden.

Die Effizienz der einzelnen Heuristiken und ihre Wechselwirkungen untereinander werden in Kapitel 7 analysiert. Die Leistungsfähigkeit der besten Kombination wird dann in Kapitel 8 mit der anderer Systeme verglichen. Die Resultate werden schließlich in Kapitel 9 zusammengefaßt.

Kapitel 2

Tableau-Algorithmen in der Wissensrepräsentation

2.1 Wissensrepräsentation

Die Wissensrepräsentation ist ein Teilgebiet der theoretischen Informatik und Künstlichen Intelligenz, deren Gegenstand die Erzeugung eines Modells für einen Teil der Realität, die *Domäne*, in einer *Wissensbasis* ist [OKvLN95]. Um diese Modellierung in einer symbolischen und maschinell auswertbaren Form zu ermöglichen, ist ein an die Domäne angepaßter Formalismus erforderlich, der im folgenden als *WR-Sprache* bezeichnet wird.

Ein Wissensrepräsentations-System enthält darüberhinaus ein System von *Inferenzregeln*, das dazu dient, aus den in der Wissensbasis explizit gespeicherten Informationen neue, implizit in diesen enthaltene abzuleiten und dem Benutzer zur Verfügung zu stellen. Dieser Mechanismus ist im *Schlußfolgerungs-Algorithmus* implementiert, der von der WR-Sprache getrennt ist. Er ist der wesentlichste Unterschied zwischen einem WR-System und einer Datenbank, die lediglich passiv die in ihr enthaltenen Fakten wiedergibt, wenn sie abgefragt werden.

Die Trennung der WR-Sprache vom Schlußfolgerungs-Algorithmus ist wichtig, um zu ermöglichen,

- daß das Modell erstellt wird, ohne daß der Benutzer über Programmiererfahrung verfügen muß,
- daß das repräsentierte Wissen jederzeit explizit als Ausdruck in der WR-Sprache und nicht nur als Teil in einem (möglicherweise großen und schwer überschaubaren) Programm vorliegt,

- daß das Modell und der Interpreter unabhängig voneinander geändert werden können, z. B. um eine effizientere Version eines Algorithmus einzusetzen, oder um die Wissensbank manuell zu modifizieren.

Die Ausgabe des Schlußfolgerungs-Algorithmus soll dabei nur von der Semantik und nicht der Syntax des repräsentierten Wissens abhängen, d. h. verschiedene syntaktische Varianten der Beschreibung desselben Sachverhalts sollen zum selben Ergebnis führen. Man spricht dann von einer *deklarativen Semantik* des Formalismus (im Gegensatz zu einer operationalen Semantik, siehe z. B. [BS01]).

2.2 Beschreibungslogiken

Eine Familie von Wissensrepräsentations-Formalismen, die die Forderung einer deklarativen Semantik erfüllt, sind die *Beschreibungslogiken* (Description Logics, DL). Sie haben eine wohldefinierte Semantik und wurden entwickelt, weil einige frühe Wissensrepräsentations-Systeme (z. B. Semantische Netze oder Frames) aufgrund des Fehlens eines formal-logischen Fundaments immer wieder unerwartete Antworten gaben. Die einfache Beschreibungslogik $\mathcal{ALC}$ („Attributive concept description Language with Complements“, [SS91]) ist wie folgt definiert (nach [BS01]):

Syntax Eine Menge $\mathcal{C}$ von *Konzeptnamen* und eine Menge $\mathcal{R}$ von *Rollennamen* seien vorgegeben. Dann ist die Menge der $\mathcal{ALC}$-*Konzeptterme* wie folgt induktiv definiert:

1. Jedes $C \in \mathcal{C} \cup \{\top, \bot\}$ ist ein Konzeptterm.
2. Sind C und D Konzeptterme und ist $r \in \mathcal{R}$, dann sind
   - $\neg C, C \sqcup D, C \sqcap D$,
   - $\forall r.C$ (*Werterestriktion*) und
   - $\exists r.C$ (*Existentielle Restriktion*)

   Konzeptterme.

($\mathcal{ALC}$)-Konzeptterme werden im folgenden auch vereinfachend als ($\mathcal{ALC}$)-*Konzepte* bezeichnet.

Semantik Eine *Interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ besteht aus einer (nichtleeren) Menge $\Delta^{\mathcal{I}}$ von Individuen und einer Funktion $\cdot^{\mathcal{I}}$, die jedes Konzept C auf eine einstellige Relation $C^{\mathcal{I}}$ und jede Rolle r auf eine zweistellige Relation $r^{\mathcal{I}}$ in $\Delta^{\mathcal{I}}$ abbildet. Die Interpretation von komplexen Konzepttermen ist wie folgt definiert:

- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
- $(\forall r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y : r^{\mathcal{I}}(x, y) \rightarrow y \in C^{\mathcal{I}}\}$
- $(\exists r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y : r^{\mathcal{I}}(x, y) \wedge y \in C^{\mathcal{I}}\}$
- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
- $\bot^{\mathcal{I}} = \emptyset$

Ein *Modell* für einen Konzeptterm C ist eine Interpretation $\mathcal{I}$, in der $C^{\mathcal{I}}$ nichtleer ist. Man sagt dann, daß die Interpretation $\mathcal{I}$ das Konzept C *erfüllt*. Ein Konzept C, für das es ein Modell gibt, heißt *erfüllbar*. Ein Konzept C *subsumiert* ein Konzept D ($C \sqsubseteq D$), wenn jedes Individuum, das C erfüllt, notwendig auch D erfüllt.

Die gewünschten Inferenz-Mechanismen für Beschreibungslogiken umfassen unter anderem Algorithmen für

- das *Subsumtionsproblem*: „Subsumiert ein Konzept C ein Konzept D?“
- das *Erfüllbarkeitsproblem*: „Gibt es eine Interpretation, die ein Konzept C erfüllt?“

In $\mathcal{ALC}$ lassen sich Subsumtion und Erfüllbarkeit wechselseitig auf einander reduzieren:

$$\begin{array}{rcl} C \sqsubseteq D & \Leftrightarrow & (C \sqcap \neg D) \text{ ist unerfüllbar} \\ C \text{ ist erfüllbar} & \Leftrightarrow & \neg(C \sqsubseteq \bot) \end{array}$$

Um Subsumtion und Erfüllbarkeit entscheiden zu können, ist es also hinreichend, einen Algorithmus für Erfüllbarkeit oder Subsumtion zu haben.

2.3 Tableau-Algorithmen

Eine häufig verwendete Art von Entscheidungsverfahren für das Erfüllbarkeitsproblem von Modal- und Beschreibungslogiken stellen die Tableau-Algorithmen dar. Ein Tableau-Algorithmus versucht, für ein gegebenes Konzept C schrittweise ein (endliches) Modell (*Tableau*) zu konstruieren, indem er nacheinander die in C enthaltenen Teilkonzepte bearbeitet.

Ein Tableau-Algorithmus für die Erfüllbarkeit eines $\mathcal{ALC}$-Konzepts ist im folgenden skizziert. Er erhält als Eingabe ein Konzept A in Negations-Normalform (mit Negation nur direkt vor Konzeptnamen) und erzeugt für A einen Baum T, in dem die Knoten mit Mengen von Konzepten und die Kanten mit Rollen beschriftet sind. Ist die Kante zwischen einem Knoten n_i und seinem Sohn n_j mit der Rolle r beschriftet, so heißt n_j *r-Nachfolger* von n_i.

Auf die in den Beschriftungen (*Labels*) enthaltenen nicht-atomaren Formeln werden die unten beschriebenen *Regeln* angewendet. Hierbei heißt eine Regel *anwendbar*, wenn durch ihre Anwendung der Baum echt verändert wird. Ein Baum heißt *vollständig*, wenn in ihm keine Regel mehr anwendbar ist, und *widerspruchsfrei*, wenn kein Knoten-Label ein Konzept C und dessen Negation $\neg C$ enthält.

- **Eingabe:** Ein Konzept A in Negations-Normalform
- Erzeuge einen Baum, der nur aus dem Wurzelknoten n_0 besteht. Beschrifte n_0 mit dem Label $\{A\}$.
- Solange eine der folgenden Regeln auf ein Konzept B im Label eines Knotens n_i anwendbar ist, wende sie an:
  - $B = C \sqcap D$: Füge C und D zum Label von n_i hinzu
  - $B = C \sqcup D$: Wähle (nichtdeterministisch) C oder D aus und füge es zum Label von n_i hinzu
  - $B = \exists r.C$: Erzeuge einen neuen r-Nachfolger n_j von n_i. Beschrifte n_j mit $\{C\}$
  - $B = \forall r.C$: Füge zu den Labels aller r-Nachfolger von n_i das Konzept C hinzu.
- **Ausgabe:** Wenn sich die Regeln so anwenden lassen, daß ein vollständiger und widerspruchsfreier Baum erzeugt wird, gebe „φ **erfüllbar**“ zurück, sonst „φ **unerfüllbar**“

$\mathcal{ALCI}$ [Spa93] ist eine Erweiterung von $\mathcal{ALC}$ um inverse Rollen (d. h. Rollen der Form r^- mit $(r^-)^{\mathcal{I}} = \{(y,x) \mid (x,y) \in r^{\mathcal{I}}\}$). Hierfür läßt sich dieser Algorithmus leicht anpassen:

- Für Konzepte der Form $\exists r^-.C$ erzeugt die $\exists$-Regel neue Nachfolger und beschriftet die neue Kante mit r^-.
- Für Konzepte der Form $\forall r.C$ und $\forall r^-.C$ fügt die $\forall$-Regel das Konzept C zu allen r-Nachbarn (Vorgängern oder Nachfolgern) hinzu, für die die Kante entsprechend beschriftet ist.

Tableau-Algorithmen sind *Entscheidungsverfahren*, d. h. sie sind korrekt, vollständig und terminierend. Die Korrektheit folgt dabei aus der Konstruktion des Tableaus, die Vollständigkeit aus der Tatsache, daß sich für eine Formel, die erfüllbar ist und für die es also ein Modell gibt, anhand dieses Modells zeigen läßt, daß eine Folge von Regelanwendungen gibt, die zur Konstruktion eines Tableaus führt.

Die Terminierung kann bei einfachen Logiken wie $\mathcal{ALC}$ daraus resultieren, daß jede Regel in einem Knoten nur einmal auf dasselbe Konzept angewendet und das neu hinzugenommene Konzept „einfacher“ ist als das ursprüngliche, der Baum endlich verzweigt ist und nie ein Knoten aus dem Baum entfernt wird. In Tableau-Algorithmen für ausdrucksstärkere Logiken ist es dagegen möglich, daß es bei der Konstruktion des Baums zu einem Zyklus kommt, in dem durch die Bearbeitung derselben Formel eine unendliche Folge von Knoten erzeugt wird. Um die Terminierung sicherzustellen, ist es dann notwendig, spezielle Techniken einzusetzen, um solche Zyklen zu entdecken und abzubrechen. Z. B. werden auf einen Knoten n nur dann Regeln angewendet, wenn es keinen Vorgänger von n gibt, dessen Label schon alle Konzepte im Label von n enthält. Man sagt dann, daß der Vorgängerknoten den Nachfolger *blockiert* [BS01].

Die Komplexität des Erfüllbarkeitsproblems ist offensichtlich abhängig von der Ausdrucksstärke der Logik, z. B. ist Erfüllbarkeit von $\mathcal{ALC}$-Konzepttermen PSPACE-vollständig [SS91]. Deshalb können Tableau-Algorithmen, die diese Probleme entscheiden, im *worst case* nicht effizienter sein. Da ihre Suche aber zielgerichtet ist und sich durch zahlreiche Heuristiken das Eintreten des worst case fast immer verhindern läßt, ist ihr Verhalten für realistische Wissensbasen vergleichbar ist mit dem unvollständiger Algorithmen, die polynomielle Laufzeit haben [Baa99]. Die Erfüllbarkeit eines Konzeptausdrucks ist dann nicht nur prinzipiell, sondern auch praktisch entscheidbar (*tractable*).

Kapitel 3

Das Bewachte Fragment der Prädikatenlogik

Das Bewachte Fragment der Prädikatenlogik ($\mathcal{GF}$) [AvBN98] wurde entwickelt, um die Ursache für die vorteilhaften modelltheoretischen Eigenschaften der Modallogik zu untersuchen (siehe Kapitel 1). Es stellte sich heraus, daß der Grund für diese Eigenschaften nicht, wie man vermuten könnte, darin begründet lag, daß sie mit einer beschränkten Zahl von Variablen (zwei) auskommen, sondern darin, daß die verwendeten Variablen in einem gewissen Sinn „eng zusammenhängen“. Was dieses formal bedeutet, wird im folgenden erklärt.

3.1 Eigenschaften von Wissensrepräsentations-Sprachen

In den meisten Anwendungen für ein Wissensrepräsentations-System möchte man die Benutzeranfragen entscheiden können. *Entscheidbarkeit* ist deshalb eine wichtige Forderung an die verwendete Sprache. Dem entgegensetzt ist die Wunsch nach hoher *Ausdrucksstärke*, denn viele ausdrucksstarke Formalismen, die als WR-Sprachen in Frage kommen, sind unentscheidbar, z. B. FO. Zwischen diesen beiden Forderungen muß also ein Kompromiß gefunden werden. Die *Komplexität* des Entscheidungsproblems sollte möglichst niedrig sein, allerdings ist schon **K** PSPACE-vollständig, und ausdrucksstärkere Logiken sind häufig in EXPTIME (z. B. PDL [FL79, Pra79]) oder NEXPTIME (z. B. FO^2, siehe Abschnitt 3.3).

Die *Modelleigenschaften* einer Logik sind dabei häufig nützlich, um (effiziente) Entscheidungsverfahren zu finden: Eine Logik hat die *Endliche-*

Modell-Eigenschaft (EME), wenn aus der Existenz eines Modells die Existenz eines endlichen Modells folgt. Um die Erfüllbarkeit einer Formel in einer solchen Logik zu testen, müssen dann nur endliche Modelle betrachtet werden. Die *Baum-Modell-Eigenschaft (BME)* gilt für eine Logik, wenn jede erfüllbare Formel ein Modell hat, dessen relationale Struktur ein Baum ist. Entsprechend müssen hier für den Erfüllbarkeitstest nur Baum-Modelle getestet werden. Für manche Logiken gilt auch die *Endliche-Baum-Modell-Eigenschaft (EBME)*, d. h. jede erfüllbare Formel hat ein endliches Baum-Modell.

3.2 Die Multimodallogik $\mathbf{K}_n$

$\mathbf{K}_n$ (siehe z. B. [BdRV01]) wurde als Erweiterung der Aussagenlogik entworfen, deren Aussagenvariablen in unterschiedlichen *Welten* unterschiedliche Werte haben können. Zwischen diesen Welten existieren *Erreichbarkeits*-Relationen R_i, mit denen die Menge der Welten eine *Kripke-Struktur* bildet. In einer bestimmten Welt können Aussagen über den Wahrheitswert von Variablen in mindestens einer über die Relation R_i erreichbaren Welt (mit dem „Diamond"-Operator $\langle i \rangle$) oder allen erreichbaren Welten (mit dem „Box"-Operator $[i]$) gemacht werden.

Man kann $\mathbf{K}_n$ auch als Fragment der Prädikatenlogik auffassen, indem man die Welten als Variablen und die in ihnen geltenden Aussagenvariablen als einstellige Prädikate beschreibt. Box- und Diamond-Operator werden dann wie folgt übersetzt:

$$\begin{array}{lcl} [i]P & \rightsquigarrow & \forall y(R_i(x,y) \rightarrow P(y)) \\ \langle i \rangle P & \rightsquigarrow & \exists y(R_i(x,y) \wedge P(y)) \end{array}$$

3.2.1 Eigenschaften von $\mathbf{K}_n$

Zu den positiven modelltheoretischen Eigenschaften von $\mathbf{K}_n$ gehört, daß es die Endliche-Baum-Modell-Eigenschaft hat. Dies ist ein Grund dafür, daß Erfüllbarkeit von $\mathbf{K}_n$-Formeln mit einem vergleichsweise einfachen Tableau-Algorithmus entscheidbar ist. Da $\mathcal{ALC}$ eine syntaktische Variante von $\mathbf{K}_n$ ist [Sch91], kann hierfür der Algorithmus aus Abschnitt 2.3 verwendet werden. Das Erfüllbarkeitsproblem für $\mathbf{K}_n$-Formeln ist PSPACE-vollständig [Lad77].

3.3 Die n-Variablen-Fragmente

Die n-Variablen-Fragmente (n *Variable Fragments*, FO^n) umfassen jeweils die Formeln der Prädikatenlogik, die höchstens n Variablen enthalten. Da man in der prädikatenlogischen Darstellung von $\mathbf{K}_n$ die Variablen x und y in den Erreichbarkeits-Relationen abwechselnd benutzen kann, ist $\mathbf{K}_n$ in FO^2 enthalten.

Die Beschränkung auf nur zwei Variablen ist stark, und daher scheint das Enthaltensein von $\mathbf{K}_n$ in FO^2 eine plausible Erklärung für die vergleichsweise niedrige Komplexität des Erfüllbarkeitsproblems für $\mathbf{K}_n$ zu sein. FO^2 ist entscheidbar [Mor75] und NExpTime-vollständig [GKV97]; es hat die Endliche-Modell-Eigenschaft [Mor75], aber die Baum-Modell-Eigenschaft und andere vorteilhafte Eigenschaften der Modallogiken fehlen ihm [AvBN98]. Zudem liefert es keinen Ansatz für weitere Erweiterungen, da FO^n für $n > 2$ unentscheidbar ist [HMT85]. Aus diesen Gründen wurde nach anderen Erweiterungen von $\mathbf{K}_n$ gesucht, die diese Nachteile nicht haben.

3.4 Die Beschränkten Fragmente und das Bewachte Fragment

Eine weitere Möglichkeit für die Erklärung der Entscheidbarkeit von $\mathbf{K}_n$ ist die Tatsache, daß die Quantifizierung „lokal“ ist in dem Sinne, daß Variablen, die gemeinsam in einer komplexen Formel vorkommen, auch „eng zusammenhängen“, d. h. gemeinsam in einem Atom vorkommen müssen. Um zu untersuchen, ob dieses der Grund für die Entscheidbarkeit ist, und wie eng der Zusammenhang mit den quantifizierten Variablen sein muß, wurden drei *Beschränkte Fragmente (Bounded Fragments, BF)* definiert, die die Stelligkeit der Relationen nicht beschränken und außerdem ein- und mehrstellige Relationen gleich behandeln [AvBN98]. Im Gegensatz hierzu können im Fragment von FO, das $\mathbf{K}_n$ entspricht (siehe Abschnitt 3.2), zweistellige Relationen nur unmittelbar nach einer Quantifizierung vorkommen. Die folgende Definition entspricht der in [Grä99b].

3.4.1 Syntax von $\mathcal{GF}$

Notation Im folgenden stehen φ, χ, ψ für Formeln, G, P, Q, R für Relationssymbole, c, d, e für Konstanten, x, y, z für Variablen, $\mathbf{c}$ und $\mathbf{x}$ für Tupel von Konstanten bzw. Variablen. Die Länge eines solchen Tupels wird mit

$|\mathbf{x}|$ bezeichnet. Als abkürzende Schreibweise werden diese Tupel gelegentlich auch als Menge der in ihnen enthaltenen Variablen aufgefaßt und Terme wie $x \in \mathbf{x}$ oder $\mathbf{x} \cup \mathbf{y}$ verwendet. „frei(φ)“ beizeichnet die freien Variablen der Formel φ, „$\varphi(\mathbf{x})$“ eine Formel mit freien Variablen $\mathbf{x}$, „$\varphi(\mathbf{c})$“ eine Formel, die nur die Konstanten aus $\mathbf{c}$ enthält.

Definition 3.1 (BFi, $\mathcal{GF}$) Die *Beschränkten Fragmente der Prädikatenlogik, BF1, BF2, BF3*, sind wie folgt induktiv definiert:

1. Jedes Atom $P(\mathbf{x})$ oder $x = y$ gehört zu BFi.
2. BFi ist abgeschlossen unter den Junktoren $\neg, \wedge, \vee, \rightarrow$.
3. Es sei $G(\mathbf{x}, \mathbf{y})$ ein Atom (der *Guard*) und φ eine BFi-Formel (der *Rumpf*).
   (a) Dann sind $\forall\mathbf{y}(G(\mathbf{x}, \mathbf{y}) \rightarrow \varphi(\mathbf{y}))$ und $\exists\mathbf{y}(G(\mathbf{x}, \mathbf{y}) \wedge \varphi(\mathbf{y}))$ BF1-Formeln.
   (b) Dann sind $\forall\mathbf{y}(G(\mathbf{x}, \mathbf{y}) \rightarrow \varphi(\mathbf{x}, \mathbf{y}))$ und $\exists\mathbf{y}(G(\mathbf{x}, \mathbf{y}) \wedge \varphi(\mathbf{x}, \mathbf{y}))$ BF2-Formeln.
   (c) Dann sind $\forall\mathbf{y}(G(\mathbf{x}, \mathbf{y}) \rightarrow \varphi(\mathbf{x}, \mathbf{y}, \mathbf{z}))$ und $\exists\mathbf{y}(G(\mathbf{x}, \mathbf{y}) \wedge \varphi(\mathbf{x}, \mathbf{y}, \mathbf{z}))$ BF3-Formeln.

BF2 wird auch das *Bewachte Fragment (Guarded Fragment, $\mathcal{GF}$)* genannt.

Im folgenden werden die neu quantifizierten Variablen in $\mathbf{y}$ *neue* Variablen und die (zuvor quantifizierten oder freien) Variablen in $\mathbf{x}$ *alte* Variablen genannt. ∎

Die Semantik ist die übliche der Prädikatenlogik.

3.4.2 Eigenschaften von $\mathcal{GF}$

Welches dieser Fragmente am besten geeignet ist, die Eigenschaften von $\mathbf{K}_n$ zu erklären, wird in [AvBN98] untersucht. Zunächst zeigt sich, daß die Beschränkten Fragmente tatsächlich verschieden sind:

Satz 3.1 (Andréka, v. Benthem, Némety) Die Beschränkten Fragmente bilden bezüglich der in ihnen zulässigen Formeln eine strikt aufsteigende Hierarchie. ∎

BF1 und $\mathcal{GF}$ haben die Endliche-Modell-Eigenschaft und sind daher entscheidbar, BF3 jedoch nicht[AvBN98]. Außerdem ist es möglich, für $\mathcal{GF}$ eine Baum-Modell-Eigenschaft zu zeigen, für die jedoch die gewöhnliche Definition eines Baum-Modells erweitert werden muß: In einem *k-Baum* [Grä99b] können in einem Knoten bis zu k verschiedene Konstanten existieren, und jede Konstante kann in beliebig vielen Knoten existieren, sofern diese einen Teilbaum bilden.

Definition 3.2 Eine Struktur $\mathfrak{B}$ (mit Trägermenge B und Signatur τ) ist ein *k-Baum*, falls ein Baum $T = (V, E)$ und eine Funktion $F : V \to \{X \subseteq B \mid |X| \leq k\}$ existieren, so daß

1. Für jedes τ-Atom $\alpha(\mathbf{x})$ und jedes Tupel $\mathbf{b}$ mit $\mathfrak{B} \models \alpha(\mathbf{b})$ gibt es einen Knoten $v \in V$ mit $\mathbf{b} \subseteq F(v)$ [1].

2. Für jedes $b \in B$ ist die Menge der Knoten $\{v \in V : b \in F(v)\}$ verbunden, d. h. sie induziert einen Teilbaum von T. ∎

Mit diesem Begriff eines Baums läßt sich zeigen, daß $\mathcal{GF}$ eine Baum-Modell-Eigenschaft hat [Grä99b]:

Satz 3.2 (Grädel) Es sei φ ein Satz in $\mathcal{GF}$ mit $m - 1$ Variablen. Wenn φ erfüllbar ist, dann existiert ein m-Baum $\mathfrak{B}$ mit $\mathfrak{B} \models \varphi$. ∎

$\mathcal{GF}$ ist also eine Erweiterung von $\mathbf{K}_n$, die, anders als FO^2, die Baum-Modell-Eigenschaft erhält. Dasselbe gilt für weitere Modelleigenschaften, die in [AvBN98] bewiesen werden. Dies deutet darauf hin, daß der enge Zusammenhang der quantifizierten Variablen, ihre „Bewachtheit“ durch das Guard-Atom, in der Modallogik die Ursache für diese Eigenschaften ist.

3.4.3 Komplexität von $\mathcal{GF}$

Entscheidbarkeit für $\mathcal{GF}$-Formeln ist 2-ExpTime-vollständig. Damit ist $\mathcal{GF}$ deutlich komplexer als FO^2 (NExpTime) oder $\mathbf{K}_n$ (PSpace). Die Ursache hierfür liegt aber in der unbeschränkten Stelligkeit der Relationen. Für eine Signatur mit beschränkter Stelligkeit (und damit für jede endliche Signatur) ist die Entscheidbarkeit von $\mathcal{GF}$-Formeln ExpTime-vollständig, also sogar weniger komplex als FO^2 [Grä99b].

[1] Hier beizeichnet $\mathbf{b}$ die Menge der im Tupel enthaltenen Variablen, siehe Abschnitt 3.4.1.

3.4.4 Ausdrucksstärke von $\mathcal{GF}$

Der offensichtliche Vorteil von $\mathcal{GF}$ gegenüber $\mathbf{K}_n$ und den meisten darauf aufbauenden Logiken ist die Möglichkeit, n-stellige Relationen zu verwenden. Darüberhinaus ist es möglich, *globale Axiome* auszudrücken: Dies sind Formeln, die für alle Elemente der Trägermenge gelten. In der Schreibweise der Prädikatenlogik hat ein globales Axiom der Form $\forall x \varphi(x)$. Diese Formel ist zwar nicht bewacht, aber sie kann in die $\mathcal{GF}$-Syntax übersetzt werden:

$$\forall x \varphi(x) \qquad \rightsquigarrow \qquad \forall x((x = x) \rightarrow \varphi(x))$$

Die Logik **S4** (siehe z. B. [BdRV01]) entspricht **K**, wobei nur Strukturen mit reflexiver und transitiver Erreichbarkeitsrelation betrachtet werden. Sie läßt sich ebenfalls in $\mathcal{GF}$ übersetzen [dN01].

3.5 Erweiterungen von $\mathcal{GF}$

Um weitere Formeln, die in anderen entscheidbaren Sprachen (z. B. temporalen Logiken), aber nicht in $\mathcal{GF}$ zulässig sind, in einer bewachten Logik ausdrücken zu können, wurde nach Möglichkeiten gesucht, die „Guard"-Bedingung in der Definition von $\mathcal{GF}$ weniger streng zu formulieren, ohne die Entscheidbarkeit und die Modelleigenschaften zu verlieren.

3.5.1 Schwach Bewachtes Fragment

Für den Beweis der Entscheidbarkeit von $\mathcal{GF}$ ist es nicht notwendig, daß *alle* neuen und alten Variablen gemeinsam in *einem einzigen* Guard-Atom vorkommen, sondern nur, daß *jede* neue Variable *mit jeder anderen* alten oder neuen gemeinsam in einem Atom vorkommt. Wenn dies gegeben ist, kann man auch eine Konjunktion von Atomen als Guard zulassen. Für das *Schwach Bewachte Fragment (Loosely Guarded Fragment, $\mathcal{LGF}$)* gibt es eine entsprechende, weniger strenge Guard-Bedingung [vB97] (die folgende Definition entspricht der in [Grä99b]):

Definition 3.3 ($\mathcal{LGF}$) $\mathcal{LGF}$ ist definiert wie $\mathcal{GF}$ in Definition 3.1, wobei Bedingung 3 ersetzt wird durch $3'$:

$3'$. Es sei $G(\mathbf{x}, \mathbf{y}) = G_1 \wedge \ldots \wedge G_n$ eine Konjunktion von Atomen und φ eine $\mathcal{LGF}$-Formel mit $\text{frei}(\varphi) \subseteq \text{frei}(G) = \mathbf{x} \cup \mathbf{y}$. Jede neue Variable $y \in \mathbf{y}$ komme mit jeder Variable $z \in \mathbf{x} \cup \mathbf{y}$ gemeinsam in einem G_i vor.

Dann sind $\forall \mathbf{y}(G(\mathbf{x}, \mathbf{y}) \rightarrow \varphi(\mathbf{x}, \mathbf{y}))$ und $\exists \mathbf{y}(G(\mathbf{x}, \mathbf{y}) \wedge \varphi(\mathbf{x}, \mathbf{y}))$ in $\mathcal{LGF}$. ∎

Durch diese Erweiterung lassen sich weitere Formeln, etwa aus der temporalen Logik, in $\mathcal{LGF}$ ausdrücken [vB97]: Die Übersetzung von „φ **until** χ“ in die Prädikatenlogik lautet:

$$\exists y(x < y \wedge \chi(y) \wedge \forall z((x < z \wedge z < y) \rightarrow \varphi(z)))$$

und liegt nicht in $\mathcal{GF}$, aber in $\mathcal{LGF}$, weil die quantifizierte Variable z sowohl mit x wie auch mit y in einem Atom gemeinsam vorkommt.

3.5.2 Clique-Bewachtes Fragment

Es ist für den Beweis der Entscheidbarkeit ebenfalls nicht notwendig, daß neue Variablen, die nur im Guard, aber nicht im Rumpf φ (siehe Definition 3.3) enthalten sind, gemeinsam mit den anderen im Guard enthaltenen Variablen in einem Atom vorkommen. Wenn man die Definition in diesem Sinne abändert, erhält man das *Clique-Bewachte Fragment (Clique Guarded Fragment, $\mathcal{CGF}$)* [Grä99a].

Definition 3.4 (Clique-Formel, $\mathcal{CGF}$) Eine Formel α ist eine *Clique-Formel* für eine Menge $\mathbf{x} \subseteq \text{free}(\alpha)$, wenn α eine Konjunktion von Atomen ist, so daß

1. je zwei verschiedene Elemente aus $\mathbf{x}$ zusammen in einem Atom aus α enthalten sind,
2. jedes Atom mindestens ein Element aus $\mathbf{x}$ enthält,
3. jedes Element aus $\text{free}(\alpha) \setminus \mathbf{x}$ genau einmal in α vorkommt.

$\mathcal{CGF}$ ist definiert wie $\mathcal{GF}$ in Definition 3.1, wobei Bedingung 3 ersetzt wird durch $3''$:

$3''$. Es sei $\alpha(\mathbf{x}, \mathbf{y}, \mathbf{z})$ eine Clique-Formel für $\mathbf{x} \cup \mathbf{y}$ und $\varphi(\mathbf{x}, \mathbf{y})$ eine $\mathcal{CGF}$-Formel mit $\text{frei}(\varphi) \subseteq \mathbf{x} \cup \mathbf{y}$.

Dann sind $\forall \mathbf{yz}(\alpha(\mathbf{x}, \mathbf{y}, \mathbf{z}) \rightarrow \varphi(\mathbf{x}, \mathbf{y}))$ und $\exists \mathbf{yz}(\alpha(\mathbf{x}, \mathbf{y}, \mathbf{z}) \wedge \varphi(\mathbf{x}, \mathbf{y}))$ $\mathcal{CGF}$-Formeln. ∎

Die Clique-Bedingung bedeutet semantisch, daß die im Rumpf φ enthaltenen Variablen $\mathbf{x} \cup \mathbf{y}$ im *Gaifman-Graphen* (siehe z. B. [Grä99a]) der Trägerstruktur eine Clique bilden.

Es fällt auf, daß $\mathcal{CGF}$ syntaktisch keine Erweiterung von $\mathcal{LGF}$ ist, da bei $\mathcal{CGF}$ für alle Paare von alten Konstanten ein gemeinsames Vorkommen in

einem Atom gefordert wird, bei $\mathcal{LGF}$ jedoch nicht. Dies wirkt sich semantisch aber nur aus, wenn die entsprechenden Konstanten nicht zuvor selbst gemeinsam eingeführt worden sind, also nur für freie Variablen. Für Sätze hingegen ist $\mathcal{CGF}$ semantisch eine echte Erweiterung von $\mathcal{LGF}$.

Beispiel 3.1 Formel 3.1 ist ein $\mathcal{LGF}$-Satz, aber kein $\mathcal{CGF}$-Satz. Sie ist aber semantisch äquivalent zu Formel 3.2, in der einfach für das Paar (x, y) im Guard das Atom $R(x, y)$ wiederholt wird.

Formel 3.3 ist ein Beispiel für einen Satz, der zu $\mathcal{CGF}$, aber nicht zu $\mathcal{LGF}$ gehört: die Variablen u, v und w sind im Rumpf nicht enthalten und müssen deshalb auch nicht gemeinsam mit den anderen Variablen x, y und z in einem Atom vorkommen.

$$\forall xy(R(x,y) \rightarrow \exists z(R(x,z) \wedge R(y,z) \wedge S(x,y,z))) \tag{3.1}$$
$$\forall xy(R(x,y) \rightarrow \exists z(R(x,y) \wedge R(x,z) \wedge R(y,z) \wedge S(x,y,z))) \tag{3.2}$$
$$\forall xyz((T(x,y,u) \wedge T(x,z,v) \wedge T(y,z,w)) \rightarrow S(x,y,z)) \tag{3.3}$$

■

3.5.3 Fixpunkte

$\mathcal{GF}$, $\mathcal{LGF}$ und $\mathcal{CGF}$ lassen sich zusätzlich um Fixpunkt-Operatoren GFP (für den größten Fixpunkt einer Relation) und LFP (für den kleinsten Fixpunkt) zu $\mu\mathcal{GF}$ usw. erweitern. Diese Logiken sind ebenfalls entscheidbar und haben die Baum-Modell-Eigenschaft [Grä99a], aber nicht die Endliche-Modell-Eigenschaft [GW99]. Da der in Kapitel 4 beschriebene Algorithmus aber keine Fixpunktoperatoren behandelt, wird hier auf diese Erweiterungen nicht näher eingegangen.

3.5.4 Eigenschaften

Alle in diesem Abschnitt beschriebenen Erweiterungen von $\mathcal{GF}$ sind, ebenso wie $\mathcal{GF}$ selbst, entscheidbar, 2-ExpTime-vollständig für beliebige Signaturen und ExpTime-vollständig für Signaturen mit beschränkter Stelligkeit [Grä99a]. Auch $\mathcal{LGF}$ und $\mathcal{CGF}$ haben die Endliche-Modell-Eigenschaft [Hod00, HO01].

3.6 $\mathcal{GF}1^-$

$\mathcal{GF}1^-$ [LST99] ist eine Restriktion von BF1. Für $\mathcal{GF}1^-$-Formeln wird zusätzlich gefordert, daß für jedes Prädikatensymbol die Parameter in zwei disjunkte Mengen eingeteilt sind. Wenn eine Formel eine Relation mit diesem Prädikatensymbol als Guard verwendet, muß einer dieser Teile aus neuen und der andere aus alten Variablen bestehen.

Definition 3.5 ($\mathcal{GF}1^-$) $\mathcal{GF}1^-$ ist definiert wie BF1 in Definition 3.1, wobei die Bedingung 3 verschärft wird zu $3'''$:

$3'''$. Für jedes Prädikatensymbol P existiere eine Zweiteilung (i, j), so daß $i+j$ der Stelligkeit von P entspricht (geschrieben $P^{(i,j)}$). Es sei $|\mathbf{x}| = i$, $|\mathbf{y}| = j$, $\mathbf{x} \cap \mathbf{y} = \emptyset$, $\varphi(\mathbf{x})$ eine $\mathcal{GF}1^-$-Formel mit freien Variablen $\mathbf{x}$.

Dann sind

$$\begin{array}{ll} \exists \mathbf{x} P^{(i,j)}(\mathbf{x}, \mathbf{y}) \wedge \varphi(\mathbf{x}), & \exists \mathbf{x} Q^{(j,i)}(\mathbf{y}, \mathbf{x}) \wedge \varphi(\mathbf{x}), \\ \forall \mathbf{x} P^{(i,j)}(\mathbf{x}, \mathbf{y}) \rightarrow \varphi(\mathbf{x}), & \forall \mathbf{x} Q^{(j,i)}(\mathbf{y}, \mathbf{x}) \rightarrow \varphi(\mathbf{x}) \end{array}$$

in $\mathcal{GF}1^-$. ∎

$\mathcal{ALCI}$ läßt sich in $\mathcal{GF}1^-$ übersetzen [LST99]. Dabei entstehen nur ein- und zweistellige Relationen; für letztere ist die Zweiteilung stets $(1, 1)$. Auch bei den Modelleigenschaften gibt es Gemeinsamkeiten zwischen $\mathcal{GF}1^-$ und $\mathcal{ALC}(\mathcal{I})$: Die Zweiteilungs-Bedingung führt dazu, daß $\mathcal{GF}1^-$ eine Baum-Modell-Eigenschaft mit einem einfacheren Baum-Modell als dem k-Baum besitzt. Um das „Standard"-Baummodell für $\mathcal{ALC}$ (siehe Abschnitt 2.3) anzupassen, genügt es, die Knoten statt mit einzelnen Konstanten mit Tupeln von Konstanten und den für diese geltenden Formeln zu beschriften. Anders als bei BF1 oder $\mathcal{GF}$ kann eine Konstante in höchstens zwei Knoten vorkommen. Damit hat $\mathcal{GF}1^-$ sogar die *Endliche*-Baum-Modell-Eigenschaft, und die Erfüllbarkeit ist, ebenfalls wie bei $\mathcal{ALC}/\mathbf{K}_n$, PSPACE-vollständig.

Für $\mathcal{GF}1^-$ existiert die Implementierung GOST (**G**F **O**ne Minus **S**atisfiability **T**ester) [Hla00], die in Kapitel 8 mit SAGA verglichen wird, um zu testen, ob sich die unterschiedliche Komplexität der zugrundeliegenden Logiken (PSPACE gegenüber EXPTIME) auch beim Entscheiden von Benchmark-Formeln für die Logiken $\mathbf{K}$ und $\mathbf{K}^-$ ($\mathbf{K}$ mit inverser Modalität) auswirkt.

3.7 Übersicht

Abschließend werden die Eigenschaften der beschriebenen Logiken noch einmal zusammengefaßt: Abbildung 3.1 zeigt die Hierarchie der beschriebenen Logiken bezüglich der Ausdrucksstärke und die zugehörigen Komplexitätsklassen. Für Signaturen mit beschränkter Stelligkeit ist die Komplexität von $\mathcal{GF}$ und den entsprechenden Erweiterungen ExpTime statt 2-ExpTime. Die Komplexität von BF1 ist unseres Wissens bisher nicht untersucht. BF1 ist als Fragment von $\mathcal{GF}$ in 2-ExpTime enthalten und ExpTime-hart, da $\mathbf{K}$ mit globalen Axiomen ExpTime-hart ist und durch das in Abschnitt 3.4.4 angegebene Verfahren in BF1 übersetzt werden kann.

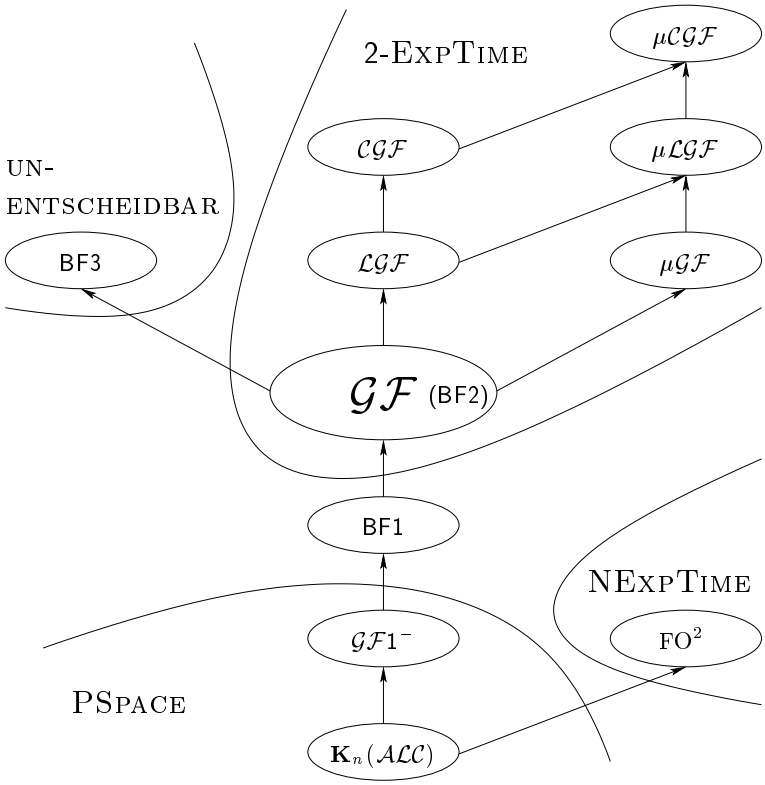

Abbildung 3.1: Hierarchie der betrachteten Logiken

Tabelle 3.1 zeigt die erwähnten Modelleigenschaften. Hier bedeuten die Abkürzungen:

- EME: Endliche-Modell-Eigenschaft,

- BME: Baum-Modell-Eigenschaft (hierbei werden für die verschiedenen Logiken verschiedene Baum-Modelle eingesetzt),
- EBME: Endliche-Baum-Modell-Eigenschaft.

Die Resultate stammen aus [AvBN98, Grä99b, Grä99a, GKV97, Hod00, HO01, LST99, Lad77, BdRV01].

Logik	Entscheidbar	EME	BME	EBME
$\mathbf{K}_n$	+	+	+	+
$\mathcal{GF}1^-$	+	+	+	+
BF1	+	+	+	-
$\mathcal{GF}$ (BF2)	+	+	+	-
$\mathcal{LGF}$	+	+	+	-
$\mathcal{CGF}$	+	+	+	-
$\mu\mathcal{GF}/\mu\mathcal{LGF}/\mu\mathcal{CGF}$	+	-	+	-
BF3	-	-	?	-

Tabelle 3.1: Eigenschaften der betrachteten Logiken

3.8 Grenzen der Ausdrucksstärke

Es gibt für verschiedene Logiken, beispielsweise für $\mathbf{K}_n$, Erweiterungen der Ausdrucksstärke. Diese erlauben es z. B., Transitivität oder Funktionalität von Erreichbarkeits-Relationen festzulegen. In FO kann man dies durch die folgenden Formeln ausdrücken:

- Funktionalität einer Relation R: $\forall xyz((Rxy \wedge Rxz) \rightarrow y = z)$
- Transitivität einer Relation R: $\forall xyz((Rxy \wedge Ryz) \rightarrow Rxz)$

Diese prädikatenlogischen Formeln sind nicht im Sinne eines der beschriebenen Fragmente bewacht. Daß es außerdem nicht möglich ist, diese Eigenschaften auf eine andere Weise in der Syntax eines der Bewachten Fragmente auszudrücken, wurde in [Grä99b] gezeigt: $\mathcal{GF}$ (mit mehr als zwei Variablen) wird unentscheidbar, wenn es um Transitivität oder Funktionalität erweitert wird.

Kapitel 4

Ein Tableau-Algorithmus für $\mathcal{GF}$

Der $\mathcal{CGF}$-Erfüllbarkeitsalgorithmus in [Grä99b] dient dazu, die Komplexität dieses Erfüllbarkeitsproblems zu bestimmen und ist nicht als Grundlage für eine Implementierung konzipiert. So arbeitet er mit Bäumen, in denen jeder Knoten für alle Instantiierungen einer Subformel der Ausgangsformel mit Konstanten dieses Knotens eine explizite Wertzuweisung „wahr“ oder „falsch“ enthält (sogenannte *Typen*). Dies führt dazu, daß für Formeln, deren Gültigkeit für die Erfüllbarkeit der Ausgangsformel irrelevant ist, nichtdeterministische Entscheidungen notwendig werden. In einem deterministischen Algorithmus muß dies durch erschöpfende Suche implementiert werden. Man kann aus diesem Grund nicht erwarten, daß das Laufzeitverhalten im *average case* signifikant besser ist als im *worst case*.

Deshalb wurde in [HT01] ein Tableau-Algorithmus für $\mathcal{CGF}$ beschrieben, der zwar im *worst case* in 2-NExpTime und damit in einer höheren Komplexitätsklasse liegt als der Algorithmus in [Grä99b], der aber die für Tableau-Algorithmen bewährten Optimierungen zuläßt und deshalb verspricht, in einer Implementierung für „realistische“ Formeln auch „praktische“ Berechenbarkeit *(Tractability)* zu erreichen. Eine genaue Untersuchung der Ausdrücke „realistisch“ und „praktisch“ würde den Rahmen dieser Arbeit sprengen. Daher werden im folgenden die in Abschnitt 7.1 beschriebenen Benchmark-Formeln als realistisch und die die auf einem gegenwärtigen Computer innerhalb des Timeouts lösbaren Probleme als praktisch entscheidbar betrachtet.

Aus demselben Grund wurde für die Implementierung Saga die Ausdrucksstärke gegenüber dem Algorithmus in [HT01] eingeschränkt: Anstelle von von $\mathcal{CGF}$ wird $\mathcal{GF}$ verwendet. Deshalb wird der Algorithmus in diesem

Kapitel in einer auf $\mathcal{GF}$ beschränkten Form wiedergegeben. Die Erweiterung von SAGA auf $\mathcal{CGF}$-Formeln ist aber ohne grundlegende Änderungen möglich.

4.1 Besonderheiten des $\mathcal{GF}$-Tableau-Algorithmus

Der $\mathcal{GF}$-Tableau-Algorithmus unterscheidet sich in wesentlichen Punkten von Tableau-Algorithmen für einfachere Logiken wie $\mathbf{K}_n/\mathcal{ALC}$ (siehe Abschnitt 2.3) oder $\mathcal{GF}1^-$ (siehe [LST99]). Die wichtigsten werden im folgenden motiviert.

4.1.1 Baum-Struktur

In Tableau-Algorithmen für Modallogiken wie $\mathbf{K}_n$ stehen die Knoten eines Baums jeweils für eine Konstante, z. B. eine Welt in einer Kripke-Struktur oder ein Individuum des Interpretationsbereichs, und die Labels enthalten die einstelligen Prädikate für diese Konstante. Die Kanten sind mit der Relation beschriftet, über die diese Konstante mit der des Vater-Knotens zusammenhängt (siehe Abschnitt 2.3). Im Unterschied hierzu erhält ein Knoten n in einem $\mathcal{GF}$-Baum, wenn er für die Erfüllung einer existentiellen Formel $\exists\mathbf{y}(G(\mathbf{c},\mathbf{y})\wedge\varphi(\mathbf{c},\mathbf{y}))$ erzeugt wird, die alten Konstanten $\mathbf{c}$ des Guard-Atoms, neue Konstanten $\mathbf{d}$ für die Variablen $\mathbf{y}$ und die Formel $G(\mathbf{c},\mathbf{d})\wedge\varphi(\mathbf{c},\mathbf{d})$. Außerdem werden die atomaren und universellen Formeln des Vater-Knotens, die nur Konstanten aus $\mathbf{c}$ enthalten, zum Label von n hinzugefügt (*propagiert*).

4.1.2 Terminierung durch Blockierung

In vielen einfacheren Logiken, etwa $\mathbf{K}_n$ oder $\mathcal{GF}1^-$, ist in jedem Knoten n die maximale Quantorentiefe der in n enthaltenen Formeln kleiner als die der im Vater enthaltenen Formeln. Die Terminierung des Algorithmus ist dann dadurch garantiert, daß auf jede Formel φ nur einmal eine Regel angewendet wird und die dabei neu hinzugenommene Formel einfacher ist als φ. In $\mathcal{GF}$ ist es dagegen leicht, eine Formel anzugeben, für die die Quantorentiefe im Sohn-Knoten nicht kleiner ist als im Vater-Knoten und die so einen unendlichen Baum erzeugen würde. Dies gilt auch, wenn $\mathcal{GF}$ ohne Gleichheit für Konstanten, d. h. ohne die Möglichkeit, globale Axiome auszudrücken, verwendet wird.

Beispiel 4.1

$$\begin{aligned}\varphi_1 &= F(c,d,e)\\ \varphi_2 &= \forall(x,y,z)(F(x,y,z) \rightarrow \varphi_3)\\ \varphi_3 &= \exists(z')(F(x,z,z') \wedge A(z'))\end{aligned}$$

Bei der Erzeugung eines Baums für $\varphi = \varphi_1 \wedge \varphi_2$ wird zunächst ein Wurzel-Knoten n_1 mit den Konstanten c, d, e und der Formel φ erzeugt. Dann wird die $\forall$-Regel auf φ_2 angewendet, und anschließend führt die $\exists$-Regel (analog zu Abschnitt 2.3) für die Formel φ_3 zur Erzeugung eines neuen Knotens n_2 mit der neuen Konstanten f. Die Konstanten c und e und die Formel φ_2, die keine Konstanten enthält, werden nach n_2 propagiert.

Hier ist die $\exists$-Regel wieder auf φ_2 anwendbar und erzeugt einen weiteren Knoten n_3 usw. In Abbildung 4.1 wird ein Tableau für diese Formel skizziert. Dabei sind in jedem Knoten zuerst die in ihm enthaltenen Konstanten und dann die für diese geltenden Formeln angegeben. Hier wird offensichtlich, daß alle neu erzeugten Knoten ähnlich sind: Sie enthalten (bis auf Umbenennung der Konstanten) dieselben Formeln. Man kann sich daher klar machen, daß es nicht notwendig ist, sie alle zu erzeugen, um die Erfüllbarkeit von φ zu überprüfen. ∎

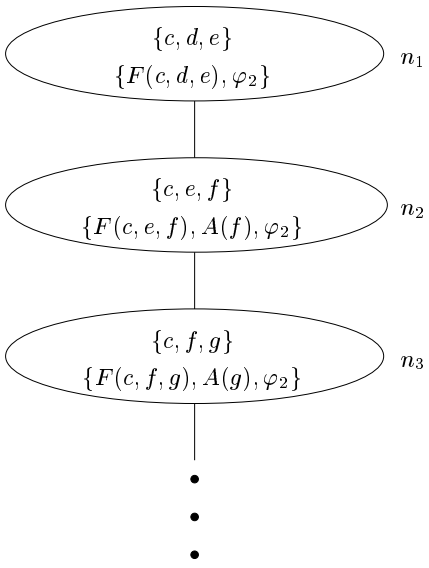

Abbildung 4.1: Ein unendliches Tableau

Um Terminierung des Algorithmus zu erreichen, ist es daher notwendig, solche Folgen von ähnlichen Knoten zu beschränken. Dies geschieht durch *Blockierung*: Für einen Knoten n werden nur dann Nachfolger erzeugt, wenn es noch keinen anderen Knoten gibt, der (für eine andere Konstantenmenge) dieselben Formeln wie n enthält. Die anschauliche Begründung, warum dieses Vorgehen einen korrekten Algorithmus liefert (d. h. keine Widersprüche übersieht), ist, daß die (Un-) Erfüllbarkeit der nicht bearbeiteten Teilformel im blockierten Knoten an einer anderen Stelle, nämlich im blockierenden Knoten, getestet wird. Wenn der Algorithmus für eine erfüllbare Formel ein (endliches) Tableau erzeugt, kann man dieses zu einem (möglicherweise unendlichen) Modell *abwickeln*, d. h. jeden blockierten Knoten durch eine Kopie des widerspruchsfreien blockierenden Knotens und ggf. seiner Nachfolger ersetzen.

4.2 Definitionen

Die Definitionen der im Tableau-Algorithmus verwendeten Begriffe werden hier nur so genau wiedergegeben, wie es für seine anschauliche Beschreibung notwendig ist. Die exakten Definitionen sind in [HT01] zu finden.

Der Algorithmus testet die Erfüllbarkeit einer Formel φ mit Hilfe eines *Vervollständigungs-Baums*: In diesem ist ein Knoten n beschriftet mit

- einer Menge $C(n)$ von Konstanten,
- einer Menge $\Delta(n)$ von Formeln, die Instantiierungen von Teilformeln von φ mit Konstanten in $C(n)$ sind,
- einer eindeutigen natürlichen Zahl $N(n)$.

Die Ordnung der Knoten im Baum ist in die numerische Ordnung der ihnen zugeordneten Zahlen eingebettet, d. h. wenn der Knoten m ein Vorgänger von n ist, so ist $N(m) < N(n)$.

Ein Knoten n heißt *direkt blockiert* von einem Knoten m, wenn

- m nicht blockiert ist,
- $N(m) < N(n)$ gilt und
- es eine injektive Abbildung π von $C(n)$ nach $C(m)$ gibt, die natürlich auf $\Delta(n)$ erweitert werden kann, so daß $\pi(\Delta(n)) = \Delta(m)|_{\pi(C(n))}$ ($\Delta(m)$ eingeschränkt auf Formeln, die nur Konstanten aus $\pi(C(n))$ enthalten) und für alle Konstanten c, die in $\Delta(m) \cap \Delta(n)$ vorkommen, gilt: $\pi(c) = c$.

Es muß also nicht nur jede Formel des blockierten Knotens im blockierenden enthalten sein, sondern (bzgl. der Konstanten in $\pi(C(n))$) auch umgekehrt. Ein Knoten ist *blockiert*, wenn er direkt blockiert ist oder wenn sein Vater blockiert ist. Da es in $\mathcal{GF}$ möglich ist, (n-stellige) inverse Rollen auszudrücken, sind diese Blockierungen *dynamisch* [HS99], d. h. sie können während der Konstruktion des Baums entstehen und später wieder aufgehoben werden, weil $\Delta(m)|_{\pi(C(n))}$ wächst.

Ein Baum T enthält einen offensichtlichen Widerspruch oder *Clash*, wenn ein Knoten eine Formel und ihre Negation oder eine Gleichheitsassertion $\neg(c \doteq c)$ enthält. Sonst heißt T *clash-frei*. T heißt *vollständig*, wenn keine Tableau-Regel (siehe 4.3) in ihm anwendbar ist.

Ein vollständiger und clash-freier Baum heißt *Tableau*.

4.3 Tableau-Regeln

$\wedge$-Regel Wenn eine Konjunktion in $\Delta(n)$ enthalten ist, aber nicht beide Konjunkte,
dann füge beide Konjunkte zu $\Delta(n)$ hinzu.

$\vee$-Regel Wenn eine Disjunktion in $\Delta(n)$ enthalten ist, aber keines der Disjunkte,
dann wähle eines der Disjunkte nichtdeterministisch aus und füge es zu $\Delta(n)$ hinzu.

$\doteq$-Regel Wenn eine Gleichheitsassertion $a \doteq b$ in $\Delta(n)$ enthalten ist für zwei Konstanten $a \neq b$,
dann ersetze in jedem Knoten m, in dem a enthalten ist, in $C(m)$ und $\Delta(m)$ a durch b.

$\forall$-Regel Wenn eine universelle Formel $\forall \mathbf{x}(G(\mathbf{c}, \mathbf{x}) \rightarrow \varphi(\mathbf{c}, \mathbf{x}))$ in $\Delta(n)$ enthalten ist,
dann füge für alle Tupel $\mathbf{d}$ von Konstanten, für die $G(\mathbf{c}, \mathbf{d})$, aber nicht $\varphi(\mathbf{c}, \mathbf{d})$ in $\Delta(n)$ enthalten ist, $\varphi(\mathbf{c}, \mathbf{d})$ zu $\Delta(n)$ hinzu.

$\exists$-Regel Wenn eine existentielle Formel $\exists \mathbf{x}(G(\mathbf{c}, \mathbf{x}) \wedge \varphi(\mathbf{c}, \mathbf{x}))$ in $\Delta(n)$ enthalten ist, n nicht blockiert ist und es weder in n noch in einem Sohn von n Konstanten $\mathbf{d}$ und Formeln $G(\mathbf{c}, \mathbf{d})$ und $\varphi(\mathbf{c}, \mathbf{d})$ gibt,
dann erzeuge einen neuen Sohn m von n mit

- einer Identifizierungsnummer $N(m)$, die größer ist als die aller bisher erzeugten Knoten,

- der Konstantenmenge $C(m) = \mathbf{c} \cup \mathbf{d}$, wobei $\mathbf{d}$ ein Tupel von neuen Konstanten für die Variablen in $\mathbf{x}$ ist,
- der Formelmenge $\Delta(m) = \{G(\mathbf{c}, \mathbf{d}) \wedge \varphi(\mathbf{c}, \mathbf{d})\}$

↕-Regel Wenn $\varphi(\mathbf{c}) \in \Delta(n)$ ein Atom oder eine universelle Formel ist, m ein Nachbar (direkter Vorgänger oder Nachfolger) von n ist und alle Konstanten aus $\mathbf{c}$, aber nicht $\varphi(\mathbf{c})$ enthält,
dann füge $\varphi(\mathbf{c})$ zu $\Delta(m)$ hinzu.

4.4 $\mathcal{GF}$-Erfüllbarkeitsalgorithmus

Die Erfüllbarkeit eines $\mathcal{GF}$-Satzes φ in Negations-Normalform wird getestet, indem ein Baum T mit einem Knoten n, einer Konstanten c, $C(n) = \{c\}$ und $\Delta(n) = \{\varphi\}$ erzeugt wird. Dann werden die Tableau-Regeln so lange angewendet, bis ein Clash auftritt oder T vollständig ist. Wenn sich die Tableau-Regeln so anwenden lassen, daß ein Tableau gefunden wird, wird „φ **erfüllbar**“ ausgegeben, sonst „φ **unerfüllbar**“.

Hierbei treten zwei unterschiedliche Arten von Nichtdeterminismus auf: Die Entscheidung, welches von zwei Disjunkten getestet wird, ist *don't-know*-nichtdeterministisch, und sie kann entscheidend für die Entdeckung eines Tableaus sein. Deshalb müssen in einer deterministischen Implementierung alle Möglichkeiten für die Auswahl von Disjunkten getestet werden (siehe Abschnitt 5.1). Demgegenüber ist die Entscheidung, welche von mehreren anwendbaren Regeln zuerst angewendet wird, *don't-care*-nichtdeterministisch, d. h. jede Entscheidung führt dazu, daß ein Tableau gefunden wird, sofern eines existiert. Die Reihenfolge der Formelanwendungen kann aber Einfluß auf das Laufzeitverhalten haben (siehe Abschnitte 6.7 und 7.3).

Satz 4.1 (Hirsch, Tobies) Der Tableau-Algorithmus entscheidet Erfüllbarkeit von $\mathcal{GF}$-Sätzen.

Beweisskizze Zu zeigen sind Terminierung, Korrektheit und Vollständigkeit. Für die ausführlichen Beweise siehe [HT01].

Terminierung Jeder Knoten n ist nur mit Instantiierungen von Subformeln der Ausgangsformel φ mit Konstanten in $C(n)$ beschriftet. Ein Knoten kann nur eine begrenzte Anzahl von Konstanten enthalten (die maximale Anzahl von quantifizierten Variablen in einer Subformel), und da die Anzahl der Subformeln endlich ist, gibt es nur endlich viele Belegungen

der Subformeln mit den Konstanten eines Knotens. Wegen der Blockierung kann deshalb nur eine beschränkte Zahl von Knoten auf einem Pfad erzeugt werden. Die Breite des Baums ist durch die Anzahl der in φ enthaltenen existentiellen Subformeln beschränkt.

Es kann auch keinen unendlichen Zyklus von Hinzunahme und Entfernung einer Formel geben, da niemals ein Knoten aus dem Baum oder eine Formel aus einem Knoten entfernt wird. Lediglich die $\doteq$-Regel entfernt Konstanten, aber dieses ist für jeden Knoten nur endlich oft möglich, weil nach der Erzeugung eines Knotens nie weitere Konstanten hinzugenommen werden.

Vollständigkeit Die $\mathcal{GF}$-Formel φ sei erfüllbar und $\mathfrak{A}$ mit Trägermenge A ein Modell von φ. Dann sind die Regeln so auf φ anwendbar, daß ein Tableau T erzeugt wird, für das eine Abbildung g von der Menge der in T enthaltenen Konstanten auf A existiert, die man natürlich auf Formeln erweitern kann, so daß für alle Knoten v in T gilt: $\mathfrak{A} \models g(\Delta(v))$. Dies läßt sich durch Induktion über die Regelanwendungen zeigen: Am Anfang gilt dies für der Funktion g mit leerem Definitionsbereich, und mit jeder Regelanwendung läßt sich g entsprechend erweitern. Da jede Folge von Regelanwendungen terminiert, wird ein Modell nach endlicher Zeit gefunden, wenn eines existiert. Weil die Funktion g existiert, ist T clash-frei, denn sonst wäre $\mathfrak{A}$ kein Modell.

Korrektheit Um aus einem für die Formel φ erzeugten Tableau T ein Modell $\mathfrak{A}$ zu konstruieren, werden die Blockierungen „abgewickelt“, d. h. Pfade, die über blockierte Knoten führen, werden ersetzt durch solche, die über den ihn blockierenden Knoten führen. Hierbei können zyklische Pfade entstehen. Zunächst wird ein Graph G erzeugt wie folgt:

1. Entferne alle indirekt blockierten Knoten aus T.

2. Erzeuge einen Graphen $G = (V_G, E_G)$, dessen Knotenmenge V_G die Menge der verbleibenden Knoten von T sind und dessen Kantenmenge E_G wie folgt definiert ist:

   - Wenn die Kante (m, n) in T enthalten und n nicht blockiert ist, ist $(m, n) \in E_G$.
   - Wenn (m, n) in T enthalten ist und n durch den Knoten n' blockiert wird, ist $(m, n') \in E_G$.

3. Erzeuge eine Menge $C(T)$ von Paaren (c, p), wobei c eine Konstante aus G und p der Pfad von der Wurzel zu dem Knoten n ist, in dem c enthalten ist. *Tail(p)* bezeichnet diesen Knoten n, den letzten Knoten des Pfades p.

4. Erzeuge eine Relation $\sim$ über $C(T)$, so daß $(a, p) \sim (b, q)$ genau dann gilt, wenn q ein Sohn von p ist und

   - $b = a$ ist, d. h. wenn die Konstante a bei der Erzeugung des Knotens $n = \text{Tail}(q)$ von dessen Vater $m = \text{Tail}(p)$ übernommen wurde, oder
   - $\text{Tail}(q)$ als Kopie für einen blockierten Knotens ein Sohn von $\text{Tail}(p)$ geworden ist und b für diese Blockierung auf a abgebildet wurde.

   Die Äquivalenzrelation $\approx$ ist der reflexiv-transitiv-symmetrische Abschluß von $\sim$.

Dann ist $A = C(T)/\approx$, die Menge der $\approx$-Restklassen.

Eine Relation $R^{\mathfrak{A}}$ über A wird wie folgt definiert: $([c_1, p]_\approx, \ldots, [c_n, p]_\approx) \in R^{\mathfrak{A}}$, wenn es in G einen Pfad p gibt, so daß $\text{Tail}(p) = m$, $c_1, \ldots, c_n \subseteq C(m)$ und $R(c_1, \ldots, c_n) \in \Delta(m)$, wenn also in G diese Relation für Elemente der entsprechenden Restklassen gilt.

Für die Struktur $\mathfrak{A}$ mit Trägermenge A und den so definierten Relationen läßt sich per Induktion über den Formelaufbau zeigen, daß diese Relationen wohldefiniert sind und $\mathfrak{A} \models \varphi$ gilt. ∎

Kapitel 5

Implementierung

Der in Kapitel 4 beschriebene Algorithmus läßt viele verschiedene Möglichkeiten zur Implementierung offen. Vor dem Beginn einer Implementierung müssen unter anderem die die folgenden Fragen beantwortet werden:

Nichtdeterminismus Die Konstruktion des Tableaus läßt sich auch als *Suche* nach einem Tableau auffassen, die bei nichtdeterministischen Entscheidungen verzweigt (*Branching*) und nach einem Clash die letzte *don't-know*-nichtdeterministische Entscheidung rückgängig macht (*Backtracking*). Beim Branching stellt sich dann die Frage: Welche Heuristiken finden möglichst schnell ein Tableau, wenn eines existiert? Wie kann der Suchraum möglichst effizient verkleinert werden, ohne die Korrektheit zu verlieren? Beim Backtracking: Wie können die Informationen zeit- und platzeffizient gespeichert werden, die notwendig sind, um nach einem Clash die Änderungen rückgängig zu machen, die infolge der letzten Verzweigung gemacht worden sind?

Datenstrukturen Welche Daten sollten zusätzlich zu den in Abschnitt 4.2 genannten in einem Knoten gespeichert werden? Welche zusätzlichen Datenstrukturen sind nötig? Welche Daten müssen global verfügbar sein?

Blockierung Wie können zwei Knoten effizient auf Blockierung getestet werden? Wie kann man die Nicht-Blockierung schnell erkennen und so den Test frühzeitig abbrechen?

5.1 Vorüberlegungen

In den folgenden Abschnitten werden die erwähnten Fragen im Detail diskutiert.

5.1.1 Nichtdeterminismus

Bei der Auswahl der nächsten zu bearbeitenden Formel oder der ersten zu testenden Alternative beim Branching führt jede faire Strategie zu einem terminierenden und korrekten Algorithmus. Die verschiedenen für diese Entscheidungen eingesetzten Heuristiken sind in den Abschnitten 6.7 und 6.5 beschrieben.

Nach einer Verzweigung und dem Clash ihrer ersten Alternative ist es notwendig, den Baum wieder in den Zustand zu versetzen, den er vor der Verzweigung hatte. Eine einfache Methode, dieses zu erreichen, die auch in GOST (siehe Abschnitt 3.6) eingesetzt wird, besteht darin, den Baum vollständig zu kopieren. Dies hat aber den Nachteil, daß für viele Knoten, die nicht als Folge der Verzweigung geändert werden, unnötig Platz (für die Kopie) und Zeit (für das Erstellen der Kopie) verbraucht wird. Dieses wird für Logiken ohne inverse Modalität, wie z. B. **K**, besonders augenfällig: Hier können die in einem Knoten n enthaltenen Formeln nur Einfluß auf die Nachfolger von n haben. Außerdem werden beim Backtracking alle Änderungen rückgängig gemacht, die *zeitlich* auf die Verzweigung *folgten*, und nicht nur die, die *logisch* von der Verzweigung *abhängen*.

Es wäre deshalb wünschenswert, alle Formeln mit den Verzweigungen, von denen sie abhängen, zu beschriften und beim Backtracking die entsprechend beschrifteten Formeln wieder zu entfernen. Leider scheitert dies an der $\doteq$-Regel, denn diese fügt nicht, wie die anderen Regeln, Formeln zu einem Knoten hinzu, sondern verändert die in einem Knoten enthaltenen Formeln. Dies könnte dadurch vermieden werden, daß die Gleichheitsrelation der Konstanten R_{eq} unabhängig von den Formeln gespeichert wird. Allerdings würde es dann notwendig, bei jedem Zugriff auf Konstanten, etwa bei der Anwendung der $\exists$-Regel oder beim Blockierungstest, die Gleichheit von zwei Formeln modulo R_{eq} zu testen, was sehr viel Zeit verbrauchen würde.

Deshalb wird in SAGA ein Mittelweg beschritten: Beim Backtracking werden Knotenkopien verwendet, aber eine Kopie wird nur dann erzeugt, wenn ein Knoten in Abhängigkeit von einer Verzweigung tatsächlich geändert wird. Diese Kopien werden in der Datenstruktur *Branching-Point* gespeichert, die bei jeder Verzweigung erzeugt wird. Um den zu einer Formel gehörenden Branching-Point zu finden, ist es notwendig, jede Formel mit

den Branching-Points zu beschriften, von denen ihr Vorhandensein im Baum abhängt. (Dieses Vorgehen scheint unverhältnismäßig aufwendig, wenn damit nur einige unnötige Knoten-Kopien eingespart werden. Der eigentliche Zweck der Beschriftung von Formeln mit Branching-Points ist das Ermöglichen von Backjumping, das in Abschnitt 6.3 beschrieben wird.)

5.1.2 Datenstrukturen

Die Knoten enthalten — zusätzlich zu Konstanten, Formeln und Identifizierungsnummer — Informationen zu bestehenden Blockierungen und den existierenden Kopien. Die Baumstruktur ist über die Identifizierungsnummern der Nachbarknoten in den einzelnen Knoten gespeichert und nicht über Zeiger. Entsprechend werden die Knoten selbst in einem Array gespeichert, so daß ihre Nummer dem Index entspricht. Dieses hat den Vorteil, daß nach der Wiederherstellung eines Knotens nach einem Clash nicht die Zeiger aller Nachbarknoten geändert werden müssen, sondern nur die Kopie an die Stelle des ursprünglichen Knotens im Array gesetzt wird.

In der Struktur Branching-Point werden neben einer Identifizierungsnummer Listen der in Abhängigkeit von der entsprechenden Verzweigung erzeugten und geänderten Knoten gespeichert. Die Branching-Points werden ebenfalls in einem Array gespeichert.

Die Formeln sind mit den Nummern der Verzweigungen beschriftet, von denen sie abhängen.

5.1.3 Blockierung

Der in SAGA verwendete Blockierungstest prüft, ob ein Knoten n von einem Knoten m mit niedrigerer Nummer blockiert wird, indem er versucht, zwischen *allen* Konstanten der Knoten eine Bijektion zu erzeugen, die die Formelmengen der beiden Knoten aufeinander abgebildet. Er weicht damit von dem im Kapitel 4 angegebenen Verfahren ab, in dem Blockierung auch dann gilt, wenn eine Bijektion zwischen den Konstanten von n und einer *Teilmenge* der Konstanten von m existiert.

Der Grund für die Modifikation ist, daß der Blockierungstest sehr aufwendig ist: Ein Knoten ist blockiert, wenn er selbst oder einer seiner Vorgänger direkt blockiert ist. Es muß also bei jeder Anwendung der $\exists$-Regel auf eine Formel im Knoten n jeder Vorgänger von n auf Blockierung getestet werden. In einem (idealerweise) balancierten binären Baum mit t Knoten hat ein Knoten $\mathcal{O}(\log t)$ Vorgänger. Jeder von diesen muß mit allen Knoten mit kleinerer Identifizierungsnummer verglichen werden, die Anzahl

dieser Knoten ist $\mathcal{O}(t)$. Damit sind $\mathcal{O}(t \log t)$ Blockierungstests nötig.

Beim Test, ob ein Knoten m mit c_m Konstanten einen Knoten n mit $c_n \leq c_m$ Konstanten blockiert, sind $\frac{c_m!}{(c_m - c_n)!}$ mögliche Abbildungen zu überprüfen. Wenn man stattdessen den Blockierungstest auf Knoten mit gleich vielen Variablen beschränkt, kann man den Test für $c_n < c_m$ sofort abbrechen und muß nur im Fall $c_m = c_n$ die $c_n!$ Permutationen testen. Zudem werden weitere Optimierungen möglich: Man kann auch für jede Art von Formeln (universelle, existentielle usw.) testen, ob die Anzahl der entsprechenden Formeln in m gleich groß ist wie in n und bei Ungleichheit abbrechen.

Um zu zeigen, daß der Algorithmus auch mit dieser Änderung noch terminiert, muß man nachweisen, daß keine unendlich langen Pfade entstehen. Dieses könnte nur dann passieren, wenn entlang eines Pfades unendlich viele Knoten erzeugt würden, d. h. wenn die $\exists$-Regel unendlich oft angewendet würde. Da es nur endlich viele existentielle Subformeln der Ausgangsformel gibt, muß die Regel unendlich oft auf dieselbe Formel angewendet werden. Die so erzeugten Knoten haben aber gerade gleich viele Konstanten und dieselben Formeln, und selbst wenn die Disjunktionen unterschiedlich expandiert würden, gibt es hierfür nur endlich viele Möglichkeiten.

Ein Nachteil ist aber, daß eine Blockierung möglicherweise später erkannt wird, d. h. unnötigerweise Knoten erzeugt werden. Für realistische Formeln ist der hierdurch zu erwartende zusätzliche Zeitaufwand aber gegenüber der oben beschriebenen Einsparung als gering einzuschätzen: Die Zeitersparnis tritt bei jedem Blockierungstest auf, der zusätzliche Zeitverbrauch nur in Einzelfällen. Und auch hier kann man annehmen, daß zumeist nur ein zusätzlicher Knoten erzeugt wird, bis die Blockierung erkannt wird.

Diese Annahme wird durch die Beobachtung gestützt, daß für Formeln der Logiken $\mathbf{K}$ und $\mathbf{K}^-$ ($\mathbf{K}$ mit inverser Modalität), für die der Algorithmus auch ohne Blockierung terminiert, das Aktivieren des Blockierungstests einen zum Teil deutlichen Effizienzgewinn bringt (siehe Abschnitte 7.3.1 und 7.3.2).

5.2 Notation

Die beschriebenen Funktionen werden in Pseudo-Code wiedergegeben. Um dort eine Mischung von deutschen und englischen Begriffen zu vermeiden, werden für die Bezeichner von Funktionen, Variablen und Datenstrukturen durchgehend englische Begriffe verwendet. Funktionen werden in *schräger* (slanted) Schrift gesetzt, Variablen in KAPITÄLCHEN. Kommentare, die durch das Zeichen „//“ eingeleitet werden, und informell wiedergegebe-

ne Anweisungen wie „move FORMULA from SET1 to SET2“ werden in der Standard-Schriftart gesetzt, Ausdrücke, die wörtlich im Code vorkommen, in `Courier`. Die Markierungen [1], [2] usw. verweisen auf Anmerkungen zu der entsprechende Code-Zeile im begleitenden Text. Die Namen von Funktionen und Variablen beginnen mit Kleinbuchstaben, die einzelnen Felder eines Record mit einem Großbuchstaben. Auf diese wird über ⟨Variablenname⟩.⟨Elementname⟩ zugegriffen. Der Operator für Wertzuweisungen ist „:=“.

5.3 Datenstrukturen

Da SAGA in Lisp implementiert ist, sind die Records als „Struct“ und die einzelnen Felder, soweit sie nicht-elementare Datenstrukturen enthalten, als Listen implementiert. In den folgenden Abschnitten werden diese wie Mengen behandelt, um die Lesbarkeit zu erhöhen.

5.3.1 Knoten

Da ein Tableau-Algorithmus einen Baum erzeugt, ist die wichtigste Datenstruktur die für einen Knoten dieses Baums. Sie enthält die in Kapitel 4.2 beschriebene eindeutige Identifizierungsnummer, die Nummern des Vorgängers und der direkten Nachfolger (und damit implizit die Kanten zu den Nachbarknoten), die in diesem Knoten existierenden Konstanten und die hierfür geltenden Formeln. Zusätzlich werden Informationen über Blockierungen und solche, die für das Branching und Backtracking notwendig sind, gespeichert.

NUMBER Die Identifizierungsnummer des Knotens, $\mathbf{N}(n)$ in Kapitel 4.2

CONSTANTS Die Konstanten dieses Knotens, $\mathbf{C}(n)$

OLD, NEW Die bereits bearbeiteten (alten) und noch zu bearbeitenden (neuen) Formeln für diese Konstanten, NODE.OLD $\cup$ NODE.NEW $= \mathbf{\Delta}(n)$, NODE.OLD $\cap$ NODE.NEW $= \emptyset$

FATHER Nummer des Vater-Knotens (oder $\emptyset$, falls der Knoten die Wurzel ist)

SONS Nummern der Sohn-Knoten (oder $\emptyset$, falls der Knoten ein Blatt ist)

BLOCKS Falls v andere Knoten blockiert, die Nummern dieser Knoten

BLOCKED Falls v blockiert ist, die Nummer des blockierenden Knotens

CREATED-BY Nummer des Verzweigungspunktes, in dessen Abhängigkeit dieser Knoten erzeugt wurde (siehe 5.3.2)

NEWEST-BID Nummer des neuesten Verzweigungspunktes, in dem eine Kopie dieses Knotens existiert

5.3.2 Verzweigungspunkt (Branching Point, BP)

Diese Datenstruktur enthält alle Informationen, die notwendig sind, um nach einem Clash die durch die letzte Verzweigung vorgenommenen Änderungen rückgängig zu machen. Außerdem erhalten auch Verzweigungspunkte eine eindeutige Nummer, die dazu dient, Formeln mit den Verzweigungspunkten zu beschriften, von denen ihr Vorhandensein im Tableau abhängt (siehe Kapitel 5.3.3).

NUMBER Identifizierungsnummer (Branching Identifier, *BID*)

MODIFIED Kopien aller in Abhängigkeit von diesem Verzweigungspunkt geänderten Knoten. Eine Kopie des Knotens n wird in dieser Liste des Branching-Points b gespeichert, bevor zu n eine Formel hinzugefügt wird, die mit b beschriftet ist (siehe Abschnitt 5.3.3).

CREATED Nummern aller in Abhängigkeit von diesem Verzweigungspunkt erzeugten Knoten. Die Nummer eines Knotens n wird dann in diese Liste des Branching-Points b eingetragen, wenn er bei der Anwendung der $\exists$-Regel auf eine Formel, die mit b beschriftet ist, erzeugt worden ist.

OTHER-ALTERNATIVE Die im Falle eines Clash hinzuzufügende Formel (z. B. die ursprüngliche Disjunktion ohne das getestete Disjunkt) und der Knoten, zu dem sie hinzuzufügen ist.

5.3.3 Beschriftete Formel

Die Formeln selbst sind als Listen repräsentiert. Tabelle 5.1 zeigt, wie die verschiedenen Arten von Formeln übersetzt werden.

Um die im Knoten vorgenommenen Änderungen den korrekten Verzweigungspunkten zuzuordnen, wird jede Formel mit der Menge der BIDs beschriftet, von denen ihr Vorhandensein im Tableau abhängt. Mit diesen Zahlen ist es möglich, die Kopie eines Knotens im richtigen Verzweigungspunkt abzulegen.

$\mathcal{GF}$-Syntax	SAGA-Syntax
Konstanten c_i	`(! i)`
Variablen v_i	`(? i)`
Vektoren $\mathbf{cv} =$	
$(c_1 \ldots c_m v_1 \ldots v_n)$	`((! 1) ... (! m) (? 1) ... (? n))`
$P(c_1 \ldots c_m v_1 \ldots v_n)$	`(P (! 1) ... (! m) (? 1) ... (? n))`
$c_i = c_j$	`(EQ (! i) (! j))`
$\varphi \wedge \chi$	`(AND` $\langle\varphi\rangle$ $\langle\chi\rangle$`)`
$\varphi \vee \chi$	`(OR` $\langle\varphi\rangle$ $\langle\chi\rangle$`)`
$\forall \mathbf{v}(G(\mathbf{cv}) \rightarrow \varphi(\mathbf{cv}))$	`(ALL` $\langle\mathbf{v}\rangle$ $\langle G(\mathbf{cv})\rangle$ $\langle\varphi(\mathbf{cv})\rangle$`)`
$\exists \mathbf{v}(G(\mathbf{cv}) \wedge \varphi(\mathbf{cv}))$	`(EX` $\langle\mathbf{v}\rangle$ $\langle G(\mathbf{cv})\rangle$ $\langle\varphi(\mathbf{cv})\rangle$`)`

Tabelle 5.1: SAGA-Syntax

FORMULA Die Formel selbst

DEPEND Menge der BIDs, von denen die Formel abhängt (*Dependency Set*)

5.3.4 Abbildungen

Bei der Anwendung der $\forall$- und $\exists$-Regeln wird eine Abbildung der neuen Variablen auf Konstanten benötigt. Diese wird als Liste von (v_i, c_j)-Paaren repräsentiert. Ähnlich werden beim Blockierungstest die Abbildungen zwischen den Konstanten der verschiedenen Knoten als (c_i, c_j)-Paare repräsentiert, wobei c_i eine Konstante des Knotens ist, dessen Blockierung getestet wird, und c_j eine Konstante des möglicherweise blockierenden Knotens.

5.4 Funktionen

In den folgenden Unterabschnitten werden die verwendeten Funktionen einzeln beschrieben. In Abbildung 5.1 ist ihre Aufrufstruktur dargestellt. Hier werden die in Abschnitt 5.4.11 erwähnten Hilfsfunktionen nur teilweise aufgeführt, um die Übersichtlichkeit zu erhöhen.

5.4.1 *construct-tableau*

Dies ist die Startfunktion, die als Eingabe eine $\mathcal{GF}$-Formel φ erhält und ein Tableau zurückgibt, wenn es existiert, sonst $\emptyset$. Es wird zuerst ein Knoten

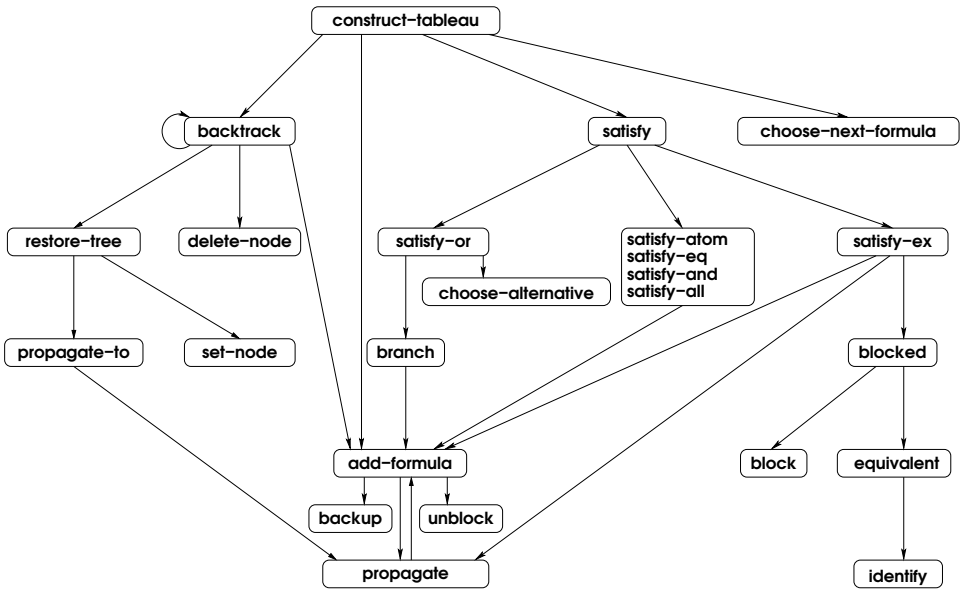

Abbildung 5.1: Aufrufstruktur der Funktionen

erzeugt, der die Formel $\langle\varphi\rangle$ und eine neue Konstante enthält [1]. $\langle\varphi\rangle$ wird dann in ihre Teilformeln zerlegt und diese der Reihe nach bearbeitet [2].

Wenn es dabei zu einem Clash kommt, wird überprüft, ob Backtracking möglich ist, d. h. ob es gegenwärtig aktive Verzweigungspunkte gibt [3]. Wenn nicht, wird $\emptyset$ zurückgegeben. Wenn alle Formeln bearbeitet worden sind, ohne daß es einen Clash gegeben hat, wird das Tableau zurückgegeben.

5.4.2 *add-formula*

Diese Funktion fügt zu einem Knoten eine neue Formel hinzu. Wenn diese bereits im Knoten enthalten ist, ist nichts zu tun [1]. Wenn ihre Negation enthalten ist, wird „`clash`“ zurückgegeben [2]. Sonst wird vor der eigentlichen Veränderung des Knotens eine Kopie in dem zu dem Dependency Set der Formel gehörigen Verzweigungspunkt gespeichert [3].

Da diese Formel den Knoten verändert, wir eine eventuelle Blockierung dieses Knotens oder eines anderen Knotens durch diesen aufgehoben [4], [5]. Hier wird deutlich, daß die Blockierungen dynamisch sind (siehe Abschnitt 4.2). Anschließend wird die Formel zur NEW-Liste hinzugefügt. Atome (einschließlich Gleichheitsassertionen) und universelle Formeln werden anschlie-

Function 1 *construct-tableau*
INPUT FORMULA
OUTPUT tableau or $\emptyset$

```
create ROOT node
CLASH := add-formula (ROOT, FORMULA)                    1
if CLASH ≠ ∅ then
  return (∅)
else
  while tree is incomplete do
    (NEXT-NODE, NEXT-FORMULA) := choose-next-formula ()
    CLASH := satisfy (NEXT-NODE, NEXT-FORMULA)          2
    if CLASH ≠ ∅ then
      if CLASH is reversible then
        backtrack ()                                    3
      else
        return (∅)
  return (ROOT)
```

ßend in die Nachbarknoten propagiert 6. Falls es hierbei zu einem Clash kommt, wird dieser zurückgegeben. Der Rückgabewert $\emptyset$ bedeutet, daß es durch das Hinzufügen der Formel nicht zu einem Clash gekommen ist.

5.4.3 *satisfy*

In Abhängigkeit von der Art der Formel (Konjunktion, Disjunktion usw.) wird die entsprechende Funktion aufgerufen. Nach dem Anwenden der entsprechenden Regel auf diese Formel wird sie aus der NEW- in die OLD-Liste verschoben.

Atome Bei Atomen wird getestet, ob der Guard einer bereits bearbeiteten universellen Formeln auf sie anwendbar ist 1. In diesem Fall wird der Rumpf der entsprechenden Formel dem Knoten hinzugefügt 2.

Gleichheits-Assertionen Jedes Vorkommen der ersten Konstanten in einer der Formeln des Knotens wird durch die zweite ersetzt 1.

Konjunktionen Beide Konjunkte werden dem Knoten hinzugefügt 1. Wenn es dabei zu einem Clash kommt, wird dieser zurückgegeben.

Function 2 *add-formula*
INPUT NODE, FORMULA
OUTPUT clash or $\emptyset$

```
if contains (NODE, FORMULA) then
  return (∅)                                                    1
else if contains (NODE, ¬FORMULA) then
  return (clash)                                                2
else
  backup (NODE, FORMULA.DEPEND)                                 3
  add FORMULA to NODE.NEW
  if NODE.BLOCKED then
    unblock (NODE)                                              4
  for all BLOCKED-NODE ∈ NODE.BLOCKS do
    unblock (BLOCKED-NODE)                                      5
  if FORMULA is atom or equality or universal then
    CLASH := propagate (NEIGHBOURS, FORMULA)                    6
    if CLASH ≠ ∅ then
      return (CLASH)
  return (∅)
```

Function 3 *satisfy-atom*
INPUT NODE, FORMULA
OUTPUT clash or $\emptyset$

```
for all OLD-FORMULA ∈ NODE.OLD do
  if OLD-FORMULA is universal then
    BINDING := match-var (guard (OLD-FORMULA), FORMULA)         1
    if BINDING ≠ fail then
      NEW-FORMULA.FORMULA :=
        substitute (body (OLD-FORMULA), BINDING)
      NEW-FORMULA.DEPEND :=
        FORMULA.DEPEND ∪ OLD-FORMULA.DEPEND
      CLASH := add-formula (NODE, NEW-FORMULA)                  2
      if CLASH ≠ ∅ then
        return (CLASH)
move FORMULA from NODE.NEW to NODE.OLD
return (∅)
```

Function 4 *satisfy-equality*
INPUT NODE, FORMULA "(`eq` CONST1 CONST2)"
OUTPUT clash or $\emptyset$

```
if CONST1 ≠ CONST2 then
  for all OLD-FORMULA ∈ NODE do
    if CONST1 ∈ constants (OLD-FORMULA) then
      NEW-FORMULA.FORMULA :=
        substitute (OLD-FORMULA, {(CONST1, CONST2)})          1
      NEW-FORMULA.DEPEND :=
        FORMULA.DEPEND ∪ OLD-FORMULA.DEPEND
      CLASH := add-formula (NODE, NEW-FORMULA)
      remove-formula (NODE, OLD-FORMULA)
      if CLASH ≠ ∅ then
        return (CLASH)
move FORMULA from NODE.NEW to NODE.OLD
return (∅)
```

Function 5 *satisfy-and*
INPUT NODE, FORMULA "(`and` CONJUNCT1 CONJUNCT2)"
OUTPUT clash or $\emptyset$

```
for all CONJUNCT ∈ {CONJUNCT1, CONJUNCT2} do
  NEW-FORMULA.FORMULA := CONJUNCT
  NEW-FORMULA.DEPEND := FORMULA.DEPEND
  CLASH := add-formula (NODE, NEW-FORMULA)                    1
  if CLASH ≠ ∅ then
    return (CLASH)
move FORMULA from NODE.NEW to NODE.OLD
return (∅)
```

Disjunktionen Eine Heuristik [1] wählt ein Disjunkt aus, das dem Knoten hinzugefügt wird. Die notwendige Sicherung der Backtracking-Informationen, wie Erzeugung eines neuen Verzweigungspunktes und Backup des Knotens, werden von der Funktion *branch* [2] durchgeführt.

Function 6 *satisfy-or*
INPUT NODE, FORMULA (`or` "DISJUNCT1 DISJUNCT2)"
OUTPUT clash or $\emptyset$

```
(ALTERNATIVE, OTHER-ALTERNATIVE) :=
    choose-alternative (DISJUNCT1, DISJUNCT2)                    [1]
CLASH := branch (NODE, ALTERNATIVE, OTHER-ALTERNATIVE)           [2]
if CLASH ≠ ∅ then
  return (CLASH)
move FORMULA from NODE.NEW to NODE.OLD
return (∅)
```

Universelle Formeln Für alle bereits bearbeiteten (alten) Atome wird getestet, ob sie eine Instantiierung des Guard sind [1]. Wenn ja, wird der Rumpf hinzugefügt [2].

Function 7 *satisfy-all*
INPUT NODE, FORMULA "(`all` VARIABLES GUARD BODY)"
OUTPUT clash or $\emptyset$

```
for all OLD-FORMULA ∈ NODE.OLD do
  if OLD-FORMULA is atom then
    BINDING := match-var (GUARD, OLD-FORMULA)                    [1]
    if BINDING ≠ fail then
      NEW-FORMULA.FORMULA := substitute (BODY, BINDING)
      NEW-FORMULA.DEPEND :=
          FORMULA.DEPEND ∪ OLD-FORMULA.DEPEND
      CLASH := add-formula (NODE, NEW-FORMULA)                   [2]
      if CLASH ≠ ∅ then
        return (CLASH)
move FORMULA from NODE.NEW to NODE.OLD
return (∅)
```

Existentielle Formeln Zunächst wird getestet, ob ein Vorgänger des Knotens (einschließlich des Knotens selbst) bereits in einem *früheren* Test

als blockiert erkannt worden ist [1]. Ist dies nicht der Fall, wird überprüft, ob eine Instantiierung der Formel (Guard und Rumpf) schon im Knoten selbst oder in einem Sohn-Knoten enthalten ist [2]. Wenn dieser Test ebenfalls fehlschlägt, wird für alle Vorgänger der Blockierungstest durchgeführt, um festzustellen, ob sie *jetzt* blockiert sind [3]. Ist auch dieses nicht gegeben, wird ein neuer Knoten mit den im Guard enthaltenen alten Konstanten und neuen Konstanten für die neuen Variablen erzeugt [4]. Guard und Rumpf werden zu diesem Knoten hinzugefügt [5]. Schließlich werden Atome (einschließlich Gleichheits-Assertionen) und universelle Formeln in den neuen Sohn propagiert [6].

5.4.4 *branch*

Ein neuer Verzweigungspunkt wird erzeugt und in den globalen Array eingetragen [1]. Das Feld OTHER-ALTERNATIVE [2] enthält die Nummer des geänderten Knotens und die Formel, die diesem Knoten hinzugefügt werden soll, wenn die erste Alternative scheitert. Dies kann (bei Syntactic Branching) die Disjunktion der verbleibenden Alternativen oder (bei Semantic Branching, siehe Abschnitt 6.2) die Negation des ersten Disjunkts[1] sein.

Schließlich wird die erste Alternative dem Knoten hinzugefügt [3]. Dabei wird (durch *add-formula*) eine Kopie des Knotens erzeugt und in die MODIFIED-Liste des Verzweigungspunktes eingetragen. Falls die Formel sofort zu einem Clash führt, wird dieser zurückgegeben.

5.4.5 *backtrack*

Die Änderungen, die infolge der letzten Verzweigung vorgenommen worden sind, werden rückgängig gemacht. Hierzu sind folgende Schritte nötig:

- Löschen der erzeugten Knoten [1]
- Ersetzen der geänderten Knoten durch ihre Backups [2]

Das weitere Vorgehen hängt davon ab, ob die erste oder zweite Alternative der Verzweigung Ursache des Clash gewesen ist. Ist es die erste Alternative (dies ist daran zu erkennen, daß das Feld OTHER-ALTERNATIVE nichtleer ist), wird zu dem entsprechenden Knoten die andere Alternative hinzugefügt

[1] Dies führt zusammen mit Boolean Constraint Propagation (siehe Abschnitt 6.4) dazu, daß später zusätzlich die Disjunktion der verbleibenden Alternativen zum Knoten hinzugefügt wird.

Function 8 *satisfy-exists*
INPUT NODE, FORMULA "(`ex` VARIABLES GUARD BODY)"
OUTPUT clash or $\emptyset$

```
for all PREDECESSOR of NODE do
  if PREDECESSOR.BLOCKED ≠ ∅ then                                  1
    return (∅)
for all C-NODE ∈ NODE ∪ NODE.SONS do
  if C-NODE contains an instance of (and GUARD BODY) then          2
    return (∅)
for all PREDECESSOR of NODE do
  if blocked (PREDECESSOR) then                                    3
    return (∅)
create SON node with new number
if FORMULA.DEPEND ≠ ∅ then
  NEWEST-BID := max (FORMULA.DEPEND)
  NEWEST-BP := get-bp (NEWEST-BID)
  SON.CREATED-BY := NEWEST-BP.NUMBER
  add SON.NUMBER to NEWEST-BP.CREATED
NEW-CONSTANTS := get-new-constants (VARIABLES)
SON.CONSTANTS := constants (GUARD) ∪ NEW-CONSTANTS               4
NEW-FORMULA.FORMULA := substitute ((and GUARD BODY),
   make-binding (VARIABLES, NEW-CONSTANTS))
NEW-FORMULA.DEPEND := FORMULA.DEPEND
CLASH := add-formula (SON, NEW-FORMULA)                          5
if CLASH ≠ ∅ then
  return (CLASH)
else
  CLASH := propagate (SON,                                       6
     atom (NODE) ∪ equality (NODE) ∪ universal (NODE))
  if CLASH ≠ ∅ then
    return (CLASH)
move FORMULA from NODE.NEW to NODE.OLD
return (∅)
```

Function 9 *branch*
INPUT NODE, ALTERNATIVE, OTHER-ALTERNATIVE
OUTPUT clash or $\emptyset$

```
NEW-BP := create-new-bp ()                                          [1]
NEW-BID := NEW-BP.NUMBER
NEW-FORMULA.FORMULA := ALTERNATIVE
NEW-FORMULA.DEPEND := {NEW-BID}
OTHER-FORMULA.FORMULA := OTHER-ALTERNATIVE
OTHER-FORMULA.DEPEND := {NEW-BID}
NEW-BP.OTHER-ALTERNATIVE.NODE := NODE
NEW-BP.OTHER-ALTERNATIVE.FORMULA := OTHER-FORMULA                   [2]
CLASH := add-formula (NODE, NEW-FORMULA)                            [3]
if CLASH ≠ ∅ then
  return (CLASH)
else
  return (∅)
```

[3]. Der Verzweigungspunkt wird durch die Funktion *reset-bp* neu initialisiert, d. h. die Felder CREATED, MODIFIED und OTHER-ALTERNATIVE werden gelöscht. Er speichert im weiteren Verlauf die Änderungen, die aus der Hinzunahme der anderen Alternative folgen.

Wenn die zweite Alternative zum Clash geführt hat (OTHER-ALTERNATIVE also leer ist), wird dieser Verzweigungspunkt gelöscht [4] und anschließend der Backtrack zum nächsten vorhergehenden Verzweigungspunkt fortgesetzt [5].

5.4.6 *backup*

In Abhängigkeit vom Dependency Set der Formel wird eine Sicherung des momentanen Zustands des Knotens durchgeführt: Der Verzweigungspunkt NEWEST-BP, in dem das Backup gespeichert werden muß, ist der neueste des Dependency Sets [1], denn die Formel hängt von allen darin enthaltenen Verzweigungspunkten ab und muß deshalb entfernt werden, sobald eine der in ihnen getroffenen Entscheidungen rückgängig gemacht wird.

Das Backup ist aber nur notwendig, wenn

1. in diesem Verzweigungspunkt noch keine Kopie dieses Knotens existiert [2] und
2. die Existenz des Knotens nicht von einem neueren Verzweigungspunkt

Function 10 *backtrack*
INPUT none
OUTPUT none

```
CURRENT-BP := get-current-bp ()
for all NODE ∈ CURRENT-BP.CREATED do
  delete-node (NODE)                                        [1]
restore-tree (CURRENT-BP.MODIFIED)                          [2]
if CURRENT-BP.OTHER-ALTERNATIVE ≠ ∅ then
  NODE := CURRENT-BP.OTHER-ALTERNATIVE.NODE
  FORMULA := CURRENT-BP.OTHER-ALTERNATIVE.FORMULA
  reset-bp (CURRENT-BP)
  CLASH := add-formula (NODE, FORMULA)                      [3]
  if CLASH ≠ ∅ then
    backtrack ()
else
  delete-bp (CURRENT-BP)                                    [4]
  backtrack ()                                              [5]
```

abhängt [3] und

3. es keinen neueren Verzweigungspunkt gibt, der ein Backup des Knotens enthält [4].

Punkt 1 ist offensichtlich: Wenn bereits eine Kopie existiert, ist keine weitere nötig. Punkt 2 benötigt eine genauere Erklärung: Wenn der Knoten in Abhängigkeit von einem neueren Verzweigungspunkt erzeugt worden ist, muß er aus dem Baum entfernt worden sein, wenn das Backtracking bis zum früheren Verzweigungspunkt fortgeschritten ist, denn es wird stets die vom neuesten Punkt abhängende Entscheidung zuerst rückgängig gemacht. Die in Abhängigkeit vom früheren Verzweigungspunkt gemachten Änderungen sind zu diesem Zeitpunkt also nicht mehr im Baum enthalten.

Beispiel 5.1 Wenn der Knoten N in Abhängigkeit vom Verzweigungspunkt 5 erzeugt wird und später in Abhängigkeit von Verzweigungspunkt 2 geändert wird, ist keine Kopie in Punkt 2 nötig, denn wenn bis zu Punkt 2 zurückgekehrt wird, ist N gar nicht mehr im Baum vorhanden (und würde, wenn er wieder eingefügt würde, zudem Formeln enthalten, die bereits entfernt worden sind, etwa die in Abhängigkeit von Punkt 5 hinzugefügten). ■

Die Begründung von Punkt 3 ist ähnlich der von Punkt 2, mit dem Unterschied, daß der Knoten vom neueren Verzweigungspunkts nicht erzeugt,

sondern geändert worden ist. Das Backup ist deswegen nicht *notwendig*, weil vor dem Wiederherstellen der Kopie in jedem Fall die Kopie des neueren Verzweigungspunkts wiederhergestellt wird, die die neue Formel ebenfalls nicht enthält, denn die Kopie wurde vorher erzeugt. Das Anlegen einer Kopie würde darüberhinaus zu einem *falschen* Verhalten des Algorithmus führen, weil die vom neueren Verzweigungspunkt vorgenommenen Änderungen in dieser Kopie enthalten wären, und somit Formeln, die nicht mehr im Baum enthalten sein sollten, beim Wiederherstellen dieser Kopie wieder in den Knoten gelangen würden.

Beispiel 5.2 Der Knoten N werde zuerst in Abhängigkeit von Verzweigungspunkt 5 und dann von Punkt 2 geändert. Wenn zu diesem Zeitpunkt in Punkt 2 eine Kopie von N erzeugt würde, enthielte sie Formeln, die von der in Punkt 5 getroffenen Entscheidung abhängen, also nicht mehr im Knoten enthalten sein sollten, wenn zu Punkt 2 zurückgekehrt wird. Das Erstellen der Kopie ist zudem überflüssig, da für den Fall, daß zu Punkt 2 zurückgekehrt werden muß, vorher notwendigerweise die Kopie aus Punkt 5 wieder in den Baum eingefügt wird. ∎

Wenn alle 3 Bedingungen erfüllt sind, wird eine vollständige Kopie von allen Feldern des Knotens erzeugt und im Verzweigungspunkt gespeichert [5]. Das Feld NEWEST-BID im Knoten wird auf den entsprechenden Verzweigungspunkt gesetzt [6].

Function 11 *backup*
INPUT NODE, DEPEND // dependency set
OUTPUT none

```
NEWEST-BID := max (FORMULA.DEPEND)                          [1]
NEWEST-BP := get-bp (NEWEST-BID)
if NODE ∉ NEWEST-BP.MODIFIED then                           [2]
  if NEWEST-BID is newer than NODE.NEWEST-BID then          [3]
    if NEWEST-BID is newer than NODE.CREATED-BY then        [4]
      BACKUP := copy (NODE)
      add BACKUP to NEWEST-BP.MODIFIED                      [5]
      NODE.NEWEST-BID := NEWEST-BID                         [6]
```

5.4.7 *propagate*

Die zu propagierende Formel wird einem Nachbarknoten hinzugefügt, wenn alle in der Formel vorkommenden Konstanten auch in diesem Knoten existieren [1]. In diesem Fall wird die entsprechende Formel auch weiter in die Nachbarknoten des neuen Knotens propagiert (durch die Funktion *add-formula*) [2].

Function 12 *propagate*
INPUT NODE-LIST, FORMULA-LIST
OUTPUT clash or $\emptyset$

```
for all FORMULA ∈ FORMULA-LIST do
  for all NODE ∈ NODE-LIST do
    if constants (FORMULA) ⊆ NODE.CONSTANTS then                [1]
      CLASH := add-formula (NODE, FORMULA)                      [2]
      if CLASH ≠ ∅ then
        return (CLASH)
return (∅)
```

5.4.8 *blocked* und Subroutinen

Um zu überprüfen, ob ein Knoten N direkt blockiert ist, wird er mit allen Knoten verglichen, die eine kleinere Nummer haben [1]. Wenn einer der Knoten M die gleiche Anzahl von Konstanten hat und für eine Bijektion zwischen den Konstanten von M und N dieselben atomaren, existententiellen und universellen Formeln enthält [2], ist N blockiert.

Function 13 *blocked*
INPUT NODE
OUTPUT `true` or `false`

```
for all OLD-NODE with OLD-NODE.NUMBER < NODE.NUMBER do        [1]
  if ¬ (OLD-NODE.BLOCKED) then
    if equivalent (NODE, OLD-NODE) then                       [2]
      block (NODE, OLD-NODE)
      return (true)
return (false)
```

equivalent Es wird überprüft, ob zwei Knoten äquivalent in dem Sinne sind, daß es eine Bijektion zwischen den Konstanten gibt, die die Prädikate,

universellen und existentiellen Formeln erhält [3] und Konstanten, die in beiden Knoten enthalten sind, auf sich selbst abbildet [2]. Voraussetzung hierfür ist, daß die Mengen der Konstanten und der entsprechenden Formeln in beiden Knoten jeweils gleich groß ist [1]. Ist dies nicht der Fall, wird der Test abgebrochen.

Function 14 *equivalent*
INPUT NODE1, NODE2
OUTPUT `true` or `false`

```
if |NODE1.CONSTANTS| = |NODE2.CONSTANTS| then                          1
  if |atom (NODE1)| = |atom (NODE2)| then                              1
    if |existential (NODE1)| = |existential (NODE2)| then              1
      if |universal (NODE1)| = |universal (NODE2)| then                1
        COMMON-CONSTANTS :=                                            2
          NODE1.CONSTANTS ∩ NODE2.CONSTANTS
        RESULT := identify (
          (atom (NODE1) ∪ existential (NODE1) ∪ universal (NODE1)),
          (atom (NODE2) ∪ existential (NODE2) ∪ universal (NODE2)),
          make-binding
            (COMMON-CONSTANTS, COMMON-CONSTANTS))                      3
        if RESULT = fail then
          return (false)
        else
          return (true)
```

identify Diese Funktion erhält als Eingabe ein Paar von Formel-Mengen und eine (partielle) Abbildung zwischen den in diesen Formeln enthaltenen Konstanten. Es wird versucht, diese Abbildung zu erweitern, indem die erste Formel FIRST-FORMULA der ersten Menge mit jeder Formel SECOND-FORMULA der zweiten Menge verglichen wird [2].

Wenn hierbei eine Formel gefunden wird, auf die FIRST-FORMULA abgebildet werden kann, wird die Funktion rekursiv mit den um die entsprechenden Formeln reduzierten Formelmengen und der ggf. auf weitere Konstanten erweiterten Abbildung aufgerufen [3]. Wenn beide Formelmengen leer sind, bedeutet dies, daß eine Bijektion gefunden wurde [1].

Function 15 *identify*
INPUT SET-PAIR, BINDINGS
OUTPUT BINDINGS or `fail`

```
if SET-PAIR = (∅, ∅) then                                              [1]
  return (BINDINGS)
else
  FIRST-SET := first (SET-PAIR)
  SECOND-SET := second (SET-PAIR)
  FIRST-FORMULA := first (FIRST-SET)
  for all SECOND-FORMULA ∈ SECOND-SET do
    NEW-BINDINGS :=
      match-const (FIRST-FORMULA, SECOND-FORMULA, BINDINGS)            [2]
    if NEW-BINDINGS ≠ fail then
      NEW-BINDINGS := identify (
        FIRST-SET \ FIRST-FORMULA,
        SECOND-SET \ SECOND-FORMULA,
        NEW-BINDINGS)                                                  [3]
      if NEW-BINDINGS ≠ fail then
        return (NEW-BINDINGS)
  return (fail)
```

5.4.9 *restore-tree*

Diese Funktion erhält als Eingabe eine Liste von Knoten-Kopien. Jede dieser Kopien wird auf die folgende Weise in den Baum eingefügt: Eventuelle Blockierungen des aktuell im Baum enthaltenen Knotens und anderer Knoten durch ihn werden zuerst aufgehoben [1], [2]. Dann wird er im Knoten-Array durch die Kopie ersetzt [4], lediglich die SON-Liste wird beibehalten [3]. (Im Vater-Knoten sind keine Änderungen nötig, da die Söhne dort nur über ihre Identifizierungsnummern gespeichert werden, siehe Abschnitt 5.1.2).

Dies verhindert, daß Knoten, deren Erzeugung unabhängig von der rückgängig zu machenden Entscheidung war, beim Backtracking gelöscht und unmittelbar danach (möglicherweise mit zahlreichen eigenen Nachfolgern) wieder erzeugt werden. Da die in Abhängigkeit von der Verzweigung erzeugten Knoten *vor* dem Aufruf von *restore-tree* entfernt werden (siehe 5.4.5), bleiben durch dieses Vorgehen auch keine Knoten fälschlich im Baum enthalten.

Beispiel 5.3 Vom Knoten n_5 werde abhängig von Verzweigung 5 eine Kopie erzeugt. Anschließend werde ein Sohn n_2 von n_5 in Abhängigkeit von Verzweigung 2 erzeugt. Wenn bei der Rückkehr zu Verzweigung 5 auch die SON-Liste der Kopie wiederhergestellt würde, würde n_2 gelöscht, obwohl er anschließend wieder erzeugt werden muß. Deshalb wird die SON-Liste beibehalten. Ein etwa in Abhängigkeit von Verzweigung 7 erzeugter Sohn n_7 wird dadurch entfernt, daß vor der Rückkehr zu Verzweigung 5 alle durch Verzweigung 7 erzeugten Knoten gelöscht werden. ∎

Es ist allerdings möglich, daß Formeln, die durch Propagieren in den Knoten gelangt sind, durch diese Art der Wiederherstellung eines Knotens verlorengehen. Deshalb muß nach dem Wiederherstellen aller Knoten erneut in diese Knoten propagiert werden.

Beispiel 5.4 Vom Knoten n werde in Verzweigungspunkt 6 eine Kopie erzeugt. Dann werde eine Formel φ, die von Verzweigung 3 abhängt, nach n propagiert. Wenn anschließend zur Verzweigung 6 zurückgekehrt wird, geht beim Wiederherstellen von n die Formel φ verloren. Deshalb muß sie neu nach n propagiert werden. ∎

Function 16 *restore-tree*
INPUT BACKUP-LIST
OUTPUT none

```
for all BACKUP ∈ BACKUP-LIST do
  NUMBER := BACKUP.NUMBER
  PRESENT-NODE := get-node (NUMBER)
  BLOCKED-NODES := PRESENT-NODE.BLOCKS
  for all NODE ∈ BLOCKED-NODES do
    unblock (NODE)                                   1
  if PRESENT-NODE.BLOCKED ≠ ∅ then
    unblock (PRESENT-NODE)                           2
  BACKUP.SONS := PRESENT-NODE.SONS                   3
  set-node (NUMBER, BACKUP)                          4
  propagate-to (BACKUP)
```

5.4.10 *choose-alternative* und *choose-next-formula*

Diese Formeln implementieren die Heuristiken für nichtdeterministische Entscheidungen: *choose-alternative* trifft die *don't-know*-Entscheidungen für die Anwendung der $\vee$-Regel, und *choose-next-formula* trifft die *don't-care*-Entscheidungen bei der Auswahl der nächsten zu bearbeitenden Formel. Es ist leicht zu sehen, daß jede faire Strategie zu einem terminierenden Algorithmus führt. Die verwendeten Heuristiken sind in Kapitel 6, ihr Einfluß auf die Geschwindigkeit des Algorithmus in Kapitel 7 beschrieben.

5.4.11 Hilfsfunktionen

Die folgenden Funktionen sind Hilfsfunktionen, die zum Zugriff auf Datenstrukturen dienen. Ihre Implementierung wird aus der Beschreibung ihrer Funktionsweise offensichtlich. Sie werden hier aufgeführt, um die Bedeutung ihrer Parameter zu verdeutlichen.

Zugriff auf Knoten

atom (NODE), *equality* (NODE), *universal* (NODE) usw. geben die Menge der im Knoten NODE enthaltenen Atome, Gleichheitsassertionen oder universellen Formeln zurück.

contains (NODE, FORMULA) testet, ob in der OLD- oder NEW-Liste des Knotens NODE die Formel FORMULA enthalten ist. Wenn ja, wird diese Formel (einschließlich ihres Dependency Sets) zurückgegeben, sonst $\emptyset$.

remove-formula (NODE, FORMULA) löscht die Formel aus dem Knoten. Dies ist nur für die Funktion *satisfy-equality* nötig, um die Formeln zu entfernen, die die ursprüngliche Konstante enthielten. In allen anderen Funktionen werden bearbeitete Formeln nicht gelöscht, sondern in die OLD-Liste verschoben.

block (NODE, PREDECESSOR) und *unblock* (NODE) tragen eine Blockierung in den Knoten ein und löschen einen solchen Eintrag. Die Felder BLOCKED im blockierten Knoten und BLOCKS in dem ihn blockierenden Vorgänger werden entsprechend modifiziert. Bei *unblock* ist die Angabe des Vorgängers nicht notwendig, da seine Nummer schon im Feld BLOCKED gespeichert ist.

copy (NODE) erzeugt eine Kopie von NODE. Dabei werden die Felder selbst kopiert, nicht nur die Referenzen auf sie. *delete-node* (NODE) löscht einen Knoten aus dem Baum. Alle Blockierungen durch ihn werden aufgehoben, und alle Söhne rekursiv gelöscht.

get-node (NODE-NUMBER) findet den im Baum enthaltenen Knoten mit der Nummer NODE-NUMBER. *set-node* (NODE-NUMBER, NODE) setzt den Knoten NODE im Baum an die Stelle des Knotens mit der Nummer NODE-NUMBER.

propagate-to (NODE) propagiert aus allen Nachbarknoten von NODE die entsprechenden Formeln in den Knoten NODE.

Zugriff auf Verzweigungspunkte

get-bp (BID) liefert den Verzweigungspunkt mit der Nummer BID zurück, *get-current-bp* () den mit der höchsten vorhandenen Nummer.

create-new-bp () erzeugt einen neuen Verzweigungspunkt, dessen Nummer um 1 größer ist als die des aktuellen. *reset-bp* (BP) löscht die Felder CREATED, MODIFIED und OTHER-ALTERNATIVE. Damit speichert der Verzweigungspunkt in Zukunft die Änderungen, die durch das Hinzufügen der zweiten Alternative entstehen. *delete-bp* (BP) löscht einen Verzweigungspunkt aus dem globalen Array.

Zugriff auf Formeln

variables (FORMULA) und *constants* (FORMULA) liefern Listen von den in FORMULA enthaltenen Variablen und Konstanten zurück. *guard* (UNIVERSAL-FORMULA) und *body* (UNIVERSAL-FORMULA) liefern für eine universelle Formel $\forall \mathbf{x}(G(\mathbf{c}, \mathbf{x}) \rightarrow \varphi(\mathbf{c}, \mathbf{x}))$ das Guard $G(\mathbf{c}, \mathbf{x})$ und den Rumpf $\varphi(\mathbf{c}, \mathbf{x})$.

match-var (FORMULA1, FORMULA2) versucht, eine Abbildung der in FORMULA1 enthaltenen Variablen auf die Konstanten von FORMULA2 zu finden, so daß die Formeln durch diese Abbildung syntaktisch gleich werden. Wenn eine solche Abbildung gefunden wird, wir sie als Liste von Paaren (siehe Abschnitt 5.3.4) zurückgegeben, sonst `fail`. *match-const* (FORMULA1, FORMULA2, BINDING) versucht, die im Parameter BINDING angegebene partielle Abbildung von Konstanten in FORMULA1 auf Konstanten in FORMULA2 so zu erweitern, daß FORMULA1 auf FORMULA2 abgebildet wird. Die Ausgabe ist analog zu *match-var*.

substitute (FORMULA, BINDING) erhält eine Formel und eine Abbildung von Konstanten oder Variablen auf Konstanten als Eingabe. Es gibt die Formel zurück, in der alle Vorkommen der im Definitionsbereich der Abbildung enthaltenen Variablen oder Konstanten durch ihre Funktionswerte ersetzt sind.

make-binding (LIST1, LIST2) erhält zwei gleich lange Listen als Eingabe und gibt eine Abbildung (siehe Abschnitt 5.3.4) zurück, wobei das n-te Element der ersten Liste auf das n-te Element der zweiten abgebildet wird.

get-new-consts (NUMBER) gibt eine Liste der Länge NUMBER zurück, die neue Konstanten enthält.

Kapitel 6

Optimierung

Kapitel 5 stellt eine möglichst einfache, relativ naive Implementierung vor. An vielen Stellen gibt es unnötige Ineffizienz. So führt z. B. die Verwendung der Negations-Normalform dazu, daß die offensichtlich unerfüllbare Formel

$$(\varphi \wedge \chi) \wedge \neg(\varphi \wedge \chi) \tag{6.1}$$

umgewandelt wird in

$$(\varphi \wedge \chi) \wedge (\neg\varphi \vee \neg\chi), \tag{6.2}$$

so daß der Widerspruch erst nach einer Verzweigung und anschließendem Backtracking erkannt wird. Analog dazu wird der Clash zwischen einer existentiellen Formel und ihrer Negation erst nach Erzeugung eines neuen Knotens erkannt. In SAGA wird eine andere Normalform verwendet, die das schnelle Erkennen von Widersprüchen unterstützt.

Darüberhinaus ist es wichtig, die Datenstrukturen, z. B. die in den Knoten enthaltenen Formeln,

- platzeffizient, so daß eine Formel, die mehrfach im Tableau vorkommt, nicht ein Mehrfaches an Speicher verbraucht, und
- zeiteffizient, so daß Gleichheit oder Widersprüchlichkeit von zwei Formeln schnell erkannt werden können,

zu verwalten.

In SAGA wurden zudem verschiedene Heuristiken für nichtdeterministische Entscheidungen implementiert, die zum Ziel haben, das Entdecken eines Tableaus (oder der Unmöglichkeit für die Existenz eines Tableaus) zu beschleunigen. Die verwendeten Heuristiken werden ebenfalls in diesem Kapitel beschrieben.

Die meisten der für SAGA verwendeten Optimierungen, insbesondere die syntaktische Vorverarbeitung, Backjumping, Semantic Branching, Boolean Constraint Propagation sowie die Heuristiken Maximize-Jump und MOMS wurden von dem DL-System FaCT [Hor97] übernommen.

6.1 Syntaktische Vorverarbeitung

Vor dem Versuch, ein Tableau zu erzeugen, ist es sinnvoll, zu untersuchen, ob die Formel oder einige ihrer Teilformeln schon aufgrund ihres syntaktischen Aufbaus trivialerweise erfüllbar oder unerfüllbar sind. Die folgenden Optimierungen dienen diesem Zweck.

6.1.1 Encoding und Lazy Unfolding

In einem Knoten werden nicht die in ihm enthaltenen Formeln selbst, sondern Code-Nummern für diese Formeln gespeichert (*Encoding*). Die Vergabe der Code-Nummern geschieht dynamisch während der syntaktischen Analyse: Beim erstmaligen Vorkommen einer Formel bekommt diese die nächste freie Nummer. Dabei erhält eine positive Formel eine gerade Zahl, ihre Negation die darauffolgende ungerade Zahl. Die Subformeln einer Formel (etwa die Disjunkte einer Disjunktion) werden in dieser Formel ihrerseits als Code-Nummern repräsentiert und numerisch sortiert. Erst wenn die Subformel selbst bearbeitet werden muß, wird sie decodiert (*Lazy Unfolding*).

Das beschleunigt einerseits das Erkennen von Tautologien (in Disjunktionen) und Widersprüchen (in Konjunktionen) innerhalb einer Formel, denn hierfür muß nur jede Zahl mit ihrem Nachbarn verglichen werden. Andererseits vereinfacht diese Normalform für Konjunktionen und Disjunktionen auch das Erkennen von Gleichheit oder Widersprüchlichkeit zwischen verschiedenen Formeln, da bei der Hinzunahme einer neuen Formel nur überprüft werden muß, ob die entsprechende Code-Nummer oder die der Negation bereits im Knoten enthalten ist. Damit dieser Test alle Äquivalenzen und Widersprüche erkennt, ist eine Normalisierung der Variablen notwendig (siehe Abschnitt 6.1.4).

Der Tableau-Algorithmus wird hierzu folgendermaßen modifiziert:

- In der Datenstruktur „Beschriftete Formel“ wird das Feld FORMULA durch das Feld NUMBER ersetzt, das die Code-Nummer der Formel enthält.
- Da das Codieren und Decodieren der Formeln effizient geschehen muß, wird für die Abbildung von Formeln auf Code-Nummern eine globa-

le Hashtabelle und für die umgekehrte Richtung ein globaler Array verwendet.

- Die Funktionen *encode* (FORMULA) und *decode* (NUMBER) bilden Formeln auf ihre Nummern bzw. Nummern auf zugehörige Formeln ab.

6.1.2 And-All-Normalform und Early Clash Detection

Um „versteckte" Clashs wie in Formel 6.2 erkennen zu können, wird eine Normalform verwendet, die die Negation von komplexen Formeln zuläßt, aber keine Disjunktionen und existentiellen Formeln verwendet. Die Umwandlung in diese Normalform geschieht mittels des DeMorgan'schen Gesetzes für Disjunktionen und unter Ausnutzung der Dualität zwischen existentieller und universeller Quantifizierung:

$$\begin{aligned} \varphi \vee \chi &\rightsquigarrow \neg(\neg\varphi \wedge \neg\chi) \\ \exists \mathbf{x}(G(\mathbf{c},\mathbf{x}) \wedge \varphi(\mathbf{c},\mathbf{x})) &\rightsquigarrow \neg(\forall \mathbf{x}(G(\mathbf{c},\mathbf{x}) \rightarrow \neg\varphi(\mathbf{c},\mathbf{x}))) \end{aligned}$$

Auf diese Weise werden Formeln, in denen triviale Clashs in versteckter Form enthalten sind, so transformiert, daß der Widerspruch erkannt wird, bevor die Formeln selbst bearbeitet werden (*Early Clash Detection, ECD*).

Darüberhinaus verwendet die Normalform nicht nur binäre, sondern n-stellige Konjunktionen (und damit implizit auch n-stellige Disjunktionen). Dies ermöglicht es, hierarchisch geschachtelte Konjunktionen in „flache", also nicht-hierarchische Äquivalente umzuwandeln. Auch dieses verbessert die Early Clash Detection, weil Clashs zwischen Formeln, die in unterschiedlichen Teilen der Konjunktion stehen, einfacher erkannt werden können. Die Teilformeln werden durch ihre Code-Nummern dargestellt und diese sortiert, so daß jede Zahl nur mit ihrem Nachbarn verglichen werden muß, um auf das Vorhandensein eines Clashs zu testen.

$$(\varphi \wedge \chi) \wedge (\psi \wedge \vartheta) \rightsquigarrow \bigwedge(2,4,6,8)$$

6.1.3 Syntaktische Vereinfachung

Nach Bildung der Normalform und Codierung der Teilformeln werden Tautologien und Clashs nach den folgenden Regeln vereinfacht:

$$\begin{array}{rcrcl} \varphi & \wedge & \varphi & \rightsquigarrow & \varphi \\ \varphi & \wedge & \neg\varphi & \rightsquigarrow & \bot \\ \varphi & \wedge & \top & \rightsquigarrow & \varphi \\ \varphi & \wedge & \bot & \rightsquigarrow & \bot \\ \forall \mathbf{x}(G(\mathbf{x},\mathbf{y}) & \rightarrow & \top) & \rightsquigarrow & \top \end{array} \qquad \begin{array}{rcrcl} \varphi & \vee & \varphi & \rightsquigarrow & \varphi \\ \varphi & \vee & \neg\varphi & \rightsquigarrow & \top \\ \varphi & \vee & \bot & \rightsquigarrow & \varphi \\ \varphi & \vee & \top & \rightsquigarrow & \top \\ \exists \mathbf{x}(G(\mathbf{x},\mathbf{y}) & \wedge & \bot) & \rightsquigarrow & \bot \end{array}$$

Dabei werden die Regeln für Disjunktionen und existentielle Formeln in der rechten Spalte nicht explizit, sondern implizit durch Umwandlung in die And-All-Normalform und Anwendung der Regeln für Konjunktionen und universelle Formeln angewendet.

Für diese Optimierung wird in der Funktion *construct-tableau* als erste Anweisung

FORMULA := *simplify* (FORMULA)

ausgeführt. Die Funktion *simplify* (FORMULA) erhält als Eingabe eine Formel, führt die beschriebenen Vereinfachungen durch und gibt die vereinfachte Formel zurück. Diese Funktion wird nicht im Pseudo-Code wiedergegeben, da dies ihre Funktionsweise nicht deutlicher machen würde als die Beschreibung in den letzten Abschnitten.

6.1.4 Normalisierung der Variablen

ECD funktioniert zusammen mit dem in Abschnitt 6.1.1 beschriebenen Encoding nur, wenn die betrachteten Formeln syntaktisch gleich sind, das heißt für universelle Formeln insbesondere, daß die quantifizierten Variablen identisch sind. Um dieses sicherzustellen, wird in der Funktion *add-formula* als erste Anweisung

FORMULA := *normalize-variables* (FORMULA)

ausgeführt. Die Funktion *normalize-variables* (FORMULA) normalisiert die in FORMULA quantifizierten Variablen so, daß in jeder Teilformel für die neuen Variablen $(x_i, x_j, x_k, \ldots)$ die niedrigsten Nummern $i = n, j = n + 1, k = n + 2, \ldots$ verwendet werden, die nicht für alte Variablen stehen. Die so normalisierte Formel wird zurückgegeben.

Beispiel 6.1 Ein abschließendes Beispiel verdeutlicht das Zusammenwirken der verschiedenen Elemente der syntaktischen Vorverarbeitung:

$(P(c) \wedge \forall x(G(c,x) \rightarrow Q(x)) \wedge \exists y(G(c,y) \wedge \neg Q(y)))$	Ausgangsformel
$(P(c) \wedge \forall x(G(c,x) \rightarrow Q(x)) \wedge \exists x(G(c,x) \wedge \neg Q(x)))$	Variablen-Norm.
$(P(c) \wedge \forall x(G(c,x) \rightarrow Q(x)) \wedge \neg(\forall x(G(c,x) \rightarrow Q(x))))$	And-All-NF
$\bigwedge(P(c), \forall x(G(c,x) \rightarrow Q(x)), \neg(\forall x(G(c,x) \rightarrow Q(x))))$	n-st. Konjunktion
$\bigwedge(2,4,5)$	Encoding
$\bot$	ECD

∎

6.2 Semantic Branching

Die naive Methode, eine Disjunktion $\chi = \varphi \vee \psi$ zu bearbeiten, ist, zunächst φ zu testen, und, falls dieses zu einem Clash führt, anschließend ψ zu testen (*Syntactic Branching*). Der Nachteil dieses Ansatzes ist, daß Zeit verbraucht worden ist, um festzustellen, daß φ (im aktuellen Baum) zu einem Clash führt, diese Information aber anschließend vergessen wird.

Die Technik des *Semantic Branching* verwendet diese Information weiter, indem sie nach dem Clash der ersten Alternative φ die Disjunktion χ im Knoten beläßt und $\neg\varphi$ hinzufügt. Dieses führt zusammen mit Boolean Constraint Propagation (siehe 6.4) dazu, daß anschließend auch ψ hinzugefügt wird. Auf diese Weise wird der Suchraum verkleinert, denn ein Baum, der φ enthält, wird nie mehr getestet. Semantic Branching beschleunigt deshalb besonders das Erkennen der Unerfüllbarkeit einer Formel.

Diese Technik kann sich aber auch nachteilig auswirken, wenn $\neg\varphi$ komplex ist, z. B. eine existentielle Formel, die weitere existentielle Formeln enthält. Dann werden neue Knoten erzeugt, Blockierungen getestet und Propagierungen in einen Knoten durchgeführt, der in einem Modell der Formel nicht enthalten sein müßte. Das Tableau wird größer als nötig. Deshalb ist es wichtig, die erste Alternative sorgfältig auszuwählen. Hierfür werden die in Kapitel 6.5 beschriebenen Heuristiken verwendet. In Kapitel 7 wird dargestellt, welcher Effekt überwiegt.

Im Algorithmus muß lediglich die Funktion *choose-alternative* so modifiziert werden, daß sie ein Paar (FORMULA, (`not` FORMULA)) zurückgibt.

6.3 Backjumping

Naives Backtracking kehrt nach einem Clash grundsätzlich zur letzten Verzweigung zurück, unabhängig davon, ob die dort getroffenene Entscheidung eine Ursache für den Clash war oder nicht. Wenn sie es nicht war, wird auch die andere Alternative zum Clash führen und erst danach zur früheren Verzweigung zurückgekehrt werden. Aber auch auf der nächst höheren Ebene ist es wieder möglich, daß die hier getroffene Entscheidung keinen Einfluß auf den Clash hatte.

Die Ineffizienz des naiven Backtracking wird beim Betrachten des folgenden Beispiels deutlich: Ein Knoten enthalte zwei Formeln φ und ψ, die notwendig zu einem Clash führen, und n zweistellige Disjunktionen. Wenn zuerst φ, dann die Disjunktionen und erst zuletzt ψ bearbeitet werden, werden 2^n verschiedene Bäume getestet, obwohl keiner von ihnen clash-frei sein

kann.

Backjumping eliminiert diese Ineffizienz, indem bei einem Clash zu der jüngsten Verzweigung zurückgekehrt wird, von der eine der am Clash beteiligten Formeln abhängt. Hierzu wird die Vereinigung der Dependency Sets dieser Formeln ermittelt und das Element mit der höchsten Identifizierungsnummer bestimmt. Dann wird diesem Verzweigungspunkt zurückgekehrt, wobei die dazwischenliegenden übersprungen werden, d. h. die von ihnen abhängigen Änderungen werden rückgängig gemacht, ohne noch die eventuell möglichen anderen Alternativen zu testen. Intuitiv ist dieses Vorgehen korrekt, da auch mit den anderen Alternativen der Clash unvermeidlich ist.

Im Algorithmus sind folgende Änderungen nötig:

- *add-formula* gibt bei einem Clash nicht nur „`clash`“ zurück, sondern das *Clash Set*, d. h. die Vereinigung der DEPEND-Felder der sich widersprechenden Formeln.

- Diese Menge wird an die Funktion *backjump* weitergegeben, die *backtrack* ersetzt.

- Die Datenstruktur „Branching Point“ erhält ein zusätzliches Feld FIRST-DEPEND, in das nach einem Clash der ersten Alternative das entsprechende Dependency Set eingetragen wird.

Von der Funktion *backjump* werden nach einem Clash die Änderungen aller Verzweigungspunkte, die zwischen dem aktuellen Verzweigungspunkt und dem jüngsten im Clash Set enthaltenen liegen [1], rückgängig gemacht. Wenn die erste Alternative Ursache des Clash war, wird das Clash Set im Feld FIRST-DEPEND des entsprechenden Verzweigungspunktes gespeichert [2]. Anschließend wird, wie bei *backtrack*, die zweite Alternative zu dem entsprechenden Knoten hinzugefügt.

Wenn die zweite Alternative zum Clash geführt hat, bedeutet dies, daß beide Alternativen im gegenwärtigen Vervollständigungsbaum notwendig zu einem Clash führen. Der Backjump muß deshalb bis zum nächsten Verzweigungspunkt fortgesetzt werden [4], von dem das Vorhandensein der einander widersprechenden Formeln abhängig ist. Das neue Clash Set ist (bei Semantic Branching) die Vereinigung der Sets der ersten und zweiten Alternative, abzüglich des aktuellen Verzweigungspunktes [3].

Function 17 *backjump*
INPUT CLASH clash set
OUTPUT none

```
NEWEST-BP := newest-bp (CLASH)
CURRENT-BP := get-current-bp ()
for all BP with
NEWEST-BP.NUMBER ≤ BP.NUMBER ≤ CURRENT-BP.NUMBER do        [1]
  for all NODE ∈ BP.CREATED do
    delete-node (NODE)
  restore-tree (BP.MODIFIED)
if NEWEST-BP.OTHER-ALTERNATIVE ≠ ∅ then
  NEWEST-BP.FIRST-DEPEND := CLASH                          [2]
  ... // as in backtrack
else
  NEW-CLASH := (CLASH ∪ BP.FIRST-DEPEND) \ NEWEST-BP       [3]
  backjump (NEW-CLASH)                                     [4]
```

6.4 Lokale Vereinfachung und Boolean Constraint Propagation

Boolean Constraint Propagation (BCP) ist eine Optimierung, die versucht, unnötige Verzweigungen zu vermeiden und so das Bearbeiten von Disjunktionen zu beschleunigen: Bevor ein Disjunkt für die Verzweigung ausgewählt wird, wird für jedes Disjunkt φ jeder im Knoten enthaltenen Disjunktionen getestet, ob es *geschlossen* ist, d. h. ob es selbst oder seine Negation $\neg\varphi$ bereits im Knoten enthalten ist, oder ob es *offen* ist. Gibt es kein offenes oder bereits im Knoten enthaltenes Disjunkt, bedeutet dies einen Clash. Gibt es nur eins, so wird dieses dem Knoten deterministisch, d. h. ohne eine Verzweigung, hinzugefügt (*Boolean Constraint Propagation*).

Die Funktion *bcp* überprüft jedes Disjunkt jeder im Knoten enthaltenen Disjunktion: Wenn es bereits im Knoten enthalten ist [1], ist für die Bearbeitung dieser Disjunktion nichts weiter zu tun, und sie wird in die Menge der bearbeiteten Formeln verschoben [2]. Wenn seine Negation im Knoten enthalten ist [3], wird es aus der Menge der Disjunkte entfernt (*Lokale Vereinfachung*) [4].

In das Dependency Set der vereinfachten Formel werden die Verzweigungspunkte der Formeln aufgenommen, die zu der Vereinfachung geführt haben [5], denn die Rücknahme einer solchen Formel bei einem Backjump

verändert die Menge der offenen Disjunkte dieser Disjunktion.

Bleibt nach der Vereinfachung kein offenes Disjunkt in einer Disjunktion übrig, wird diese der Liste CLASH-LIST hinzugefügt [6]. Gibt es nur eines, ist es BCP-LIST [7]; gibt es mehrere, ist es DISJUNCT-LIST [8]. Nachdem alle Disjunktionen bearbeitet sind, werden diese drei Listen zurückgegeben.

Function 18 *bcp*
INPUT NODE
OUTPUT (CLASH-LIST, BCP-LIST, DISJUNCTION-LIST)

```
for all FORMULA ∈ NODE.NEW do
  if FORMULA is a disjunction then
    for all DISJUNCT ∈ FORMULA do
      if contains (NODE, DISJUNCT) then                              [1]
        move FORMULA from NODE.NEW to NODE.OLD                       [2]
      else
        CONTAINED-FORMULA := contains (NODE, (not DISJUNCT))
        if CONTAINED-FORMULA ≠ ∅ then                                [3]
          remove DISJUNCT from FORMULA                               [4]
          add CONTAINED-FORMULA.DEPEND to FORMULA.DEPEND             [5]
    if number of disjuncts in FORMULA is 0 then
      add FORMULA to CLASH-LIST                                      [6]
    else if number of disjuncts in FORMULA is 1 then
      add FORMULA to BCP-LIST                                        [7]
    else
      add FORMULA to DISJUNCTION-LIST                                [8]
return (CLASH-LIST, BCP-LIST, DISJUNCTION-LIST)
```

Die Funktion *satisfy-or* wird so modifiziert, daß sie nicht eine Formel, sondern einen Knoten als Eingabe erhält. Für diesen wird zunächst die Funktion *bcp* aufgerufen [1], die für jede im Knoten enthaltene Disjunktion testet, ob sie mehrere offene Disjunkte, eines oder keines enthält. Entsprechend werden drei Listen mit den um geschlossene Disjunkte reduzierten Disjunktionen zurückgegeben. Wenn es eine Disjunktion ohne offene Disjunkte gab, gibt *satisfy-or* den entsprechenden Clash zurück [2].

Sonst werden die BCP-Kandidaten bearbeitet [3]: Das jeweils einzige offene Disjunkt wird dem Knoten deterministisch, d. h. ohne Verzweigung, hinzugefügt. Nur, wenn BCP nicht möglich ist, wird die Funktion *choose-alternative* mit den reduzierten Disjunktionen aufgerufen und anschließend eine Verzweigung durchgeführt [4]. Diese kann, anders als im bisherigen

Algorithmus, nicht sofort zu einem Clash führen, da nur offene Disjunkte Kandidaten für ALTERNATIVE und OTHER-ALTERNATIVE sind. Die Modifikationen der Funktion *choose-alternative* sind in Abschnitt 6.5 beschrieben.

Function 19 *satisfy-or* // modified
INPUT NODE
OUTPUT clash or $\emptyset$

```
(CLASH-LIST, BCP-LIST, DISJUNCTION-LIST) := bcp (NODE)                [1]
if CLASH-LIST ≠ ∅ then
  CLASH-FORMULA := first (CLASH-LIST)
  return (CLASH-FORMULA.DEPEND)                                       [2]
else if BCP-LIST ≠ ∅ then
  for all BCP-FORMULA ∈ BCP-LIST do
    CLASH := add-formula (NODE, disjunct (BCP-FORMULA))               [3]
    if CLASH ≠ ∅ then
      return (CLASH)
else
  (ALTERNATIVE, OTHER-ALTERNATIVE) :=
     choose-alternative (DISJUNCTION-LIST)
  branch (NODE, ALTERNATIVE, OTHER-ALTERNATIVE)                       [4]
return (∅)
```

6.5 Verzweigungs-Heuristiken

Bei einer Verzweigung ist die Entscheidung, welche Alternative zuerst getestet wird, von großer Bedeutung für die Effizienz des Algorithmus. Hierfür gibt es verschiedene Heuristiken, die sich schon in ihrer Zielsetzung wesentlich unterscheiden: so kann man versuchen, die Wirksamkeit von BCP zu erhöhen, indem man die Negation von möglichst vielen Disjunkten zum Knoten hinzufügt, oder man kann die Anzahl der beim Backjumping übersprungenen Verzweigungspunkte maximieren.

Die beschriebenen Heuristiken sind in der modifizierten Funktion *choose-alternative* (DISJUNCTION-LIST) implementiert, die, anders als in Abschnitt 5.4.10, nicht ein Paar von Alternativen, sondern eine Liste von Disjunktionen erhält. Aus diesen werden entsprechend der gewählten Heuristik die Formeln ALTERNATIVE und OTHER-ALTERNATIVE ausgewählt und zurückgegeben.

6.5.1 Maximize-Jump

Hier wird von allen in einem Knoten enthaltenen Disjunktionen diejenige ausgewählt, die im Fall ihres Scheiterns, also eines Clashs aller Disjunkte, den weitesten Backjump verspricht, d. h. für die der jüngste im Dependency Set enthaltene Verzweigungspunkt am ältesten ist. Diese Technik wird in [HST00] beschrieben. Die zugrundeliegende Idee ist, den Suchraum zu verkleinern, indem möglichst große Äste abgeschnitten werden.

Bei der Auswahl des Disjunkts, das zuerst dem Tableau hinzugefügt wird, wird in SAGA das syntaktisch kürzeste genommen. Die Idee hinter diesem Vorgehen ist, daß die Erfüllbarkeit einer kurzen Formel im Tableau wahrscheinlich schneller überprüfbar ist als die einer langen Formel mit zahlreichen Subformeln.

6.5.2 MOMS

MOMS [Fre95] steht für „Maximum Occurrence in formulae of Minimum Size“. Diese Heuristik betrachtet die Multimenge M der Disjunkte *aller* Disjunktionen von minimaler Länge und zählt die positiven und negierten Vorkommen der einzelnen Formeln. Aus allen Disjunkten wird dasjenige φ ausgewählt, für das die Summe der Vorkommen von φ und $\neg\varphi$ in M maximal ist. Wenn die Anzahl der φ größer ist als die der $\neg\varphi$, wird zuerst $\neg\varphi$ getestet und im Fall eines Clash anschließend φ.

Dieses Vorgehen soll dazu dienen, den Effekt von BCP zu maximieren, indem möglichst viele Disjunkte aus den Disjunktionen entfernt werden und schon ab einem frühen Zeitpunkt kein Verzweigen mehr notwendig ist, sondern der Baum deterministisch expandiert wird. Die zugrundeliegende Idee läßt sich beschreiben mit „Die beste Disjunktion ist eine, die höchstens eine offene Alternative enthält.“ Diese aus der Aussagenlogik übernommene Technik hat aber verschiedene Nachteile:

- Sie nimmt keine Rücksicht auf Backjumping und kann deswegen den Algorithmus verlangsamen [HST00].

- Es wird zuerst die Alternative getestet, die weniger offene Disjunkte enthält. Dies heißt aber auch, daß zuerst die Alternative getestet wird, in der ein Clash wahrscheinlicher ist. So kommt es, besonders bei erfüllbaren Formeln, zu Zeitverlust durch häufigere Clashs und anschließende Backjumps.

6.5.3 Inverses MOMS

Im Gegensatz zu MOMS soll inverses MOMS für erfüllbare Formeln das Finden eines Tableaus beschleunigen: Anstatt zuerst die stärker beschränkte Alternative zu testen, wird die Formel hinzugefügt, die *häufiger* in M enthalten ist [Hla00]. Die Heuristik ersetzt also das Prinzip von MOMS durch „Die beste Disjunktion ist eine erfüllte Disjunktion“. Für GOST (siehe Abschnitt 3.6) war iMOMS deutlich schneller als MOMS und auch etwas schneller als Maximize-Jump.

6.6 Todo-Liste

Durch die $\updownarrow$-Regel ist es möglich, daß Formeln zum Vorgänger eines Knotens propagiert werden. Im Gegensatz etwa zu $\mathcal{ALC}$-Tableaus, in denen ein einmal vollständig expandierter Knoten nie mehr betrachtet werden muß, ist es für $\mathcal{GF}$ deswegen nötig, einen Überblick über die im Baum enthaltenen unbearbeiteten Formeln zu behalten. Um möglichst effizient, d. h. ohne Durchsuchen des gesamten Baums, die nächste zu expandierende Formel zu finden, wird die Datenstruktur „Todo-Liste“ verwendet. Sie enthält für jede Art von Formeln (Atome, Konjunktionen, Disjunktionen usw.) eine Liste aller nicht blockierten Knoten, die noch unbearbeitete Formeln dieser Art enthalten. Diese Listen sind nach den Nummern der Knoten sortiert, so daß sichergestellt ist, daß erst dann eine Formel in einem Knoten expandiert wird, wenn in keinem seiner Vorgänger eine gleichartige Formel noch nicht expandiert ist. Das Expandieren eines tieferen Knotens ist z. B. dann unnötig, wenn nach dem Expandieren eines Vorgängers seine Blockierung erkennbar wird oder der Knoten selbst als Folge eines Clashs entfernt wird.

Die in der Todo-Liste enthaltene Information ist redundant, da sie auch im Tableau enthalten ist. Sie ermöglicht aber ein schnelles Finden der nächsten zu bearbeitenden Formel.

Um die Todo-Liste konsistent zu halten, sind folgende Schritte notwendig:

- Beim Hinzufügen einer Formel zu einem Knoten wird der Knoten in die der Formel entsprechende Todo-Liste eingefügt.
- Beim Blockieren eines Knotens wird der durch ihn induzierte Teilbaum aus der Todo-Liste entfernt.
- Entsprechend wird bei der Aufhebung einer Blockierung der Teilbaum

nach unbearbeiteten Formeln durchsucht und werden die entsprechenden Knoten in die Todo-Liste eingefügt.

- Nach einem Backjump werden die wiederhergestellten Knoten in die entsprechenenden Listen eingefügt, wenn sie nicht indirekt blockiert sind.

- Ebenfalls nach einem Backjump werden die gelöschten Knoten aus der Liste entfernt.

Es ist möglich, daß in der Todo-Liste Knoten enthalten sind, die keine unbearbeiteten Formeln enthalten, etwa weil diese Formeln durch einen Backjump aus dem Knoten entfernt wurden. Dieser Fall wird von der Funktion *choose-next-formula* abgefangen, die vor dem Zurückgeben des Knotens verifiziert, daß er unbearbeitete Formeln enthält. Dieses Vorgehen ist effizienter als die Sicherstellung, daß die Todo-Liste stets konsistent ist.

Es ist aber garantiert, daß keine Knoten in der Todo-Liste fehlen, die nicht blockiert sind und unbearbeitete Formeln enthalten.

6.7 Reihenfolge der Formeln

Je einfacher die Anwendung der entsprechenden Regel auf eine Formel ist, um so früher sollte sie bearbeitet werden, damit nach einem möglichen Clash nur einfache Formeln rückgängig gemacht werden müssen und nur die Zeit für die Bearbeitung einfacher Formeln verloren ist. Die für die Anwendung der verschiedenen Regeln notwendigen Schritte sind in Tabelle 6.1 zusammengefaßt.

Atome	Matching mit universellen Formeln, Propagierung
Gleichheitsassertionen	Identifizierung von Konstanten, Propagierung
Konjunktionen	Hinzufügen der Konjunkte
Disjunktionen	Branching
Universelle Formeln	Matching mit Atomen, Propagierung
Existentielle Formeln	Blockierungstest, Erzeugen eines Knotens, Propagierung in diesen Knoten

Tabelle 6.1: Zur Bearbeitung einer Formel notwendige Schritte

Besonders einfach sind in dieser Hinsicht Atome und Konjunktionen, besonders schwer Disjunktionen und existentielle Formeln. Vielversprechend

sind also folgende zwei Reihenfolgen, die sich lediglich in den beiden letzten Positionen unterscheiden:

1. Atome ▷ Konjunktionen ▷ Gleichheitsassertionen ▷
   Universelle Formeln ▷ Existentielle Formeln ▷ Disjunktionen

2. Atome ▷ Konjunktionen ▷ Gleichheitsassertionen ▷
   Universelle Formeln ▷ Disjunktionen ▷ Existentielle Formeln

Um Variante 2 testen zu können, muß der Blockierungstest geändert werden: In der in Abschnitt 5.4.8 angegebenen Form testet er nur Äquivalenz von Atomen, universellen und existentiellen Formeln und setzt damit implizit voraus, daß alle Gleichheitsassertionen, Konjunktionen und Disjunktionen bereits bearbeitet sind. Der Grund hierfür ist, daß der auf die weiteren Formeln ausgedehnte Test Blockierung zu spät oder gar nicht erkennen würde, denn es ist irrelevant, ob eine Formel φ deterministisch oder durch eine Disjunktion $\varphi \vee \chi$ in den Knoten gelangt ist. Wenn aber der blockierte Knoten noch unbearbeitete Disjunktionen enthält, ist es möglich, daß in ihnen ein Widerspruch enthalten ist, der nicht entdeckt wird.

Deswegen ist ein Kompromiß zwischen der Effizienz des Blockierungstests und der Vermeidung von unnötigem Verzweigen notwendig: der Blockierungstest wird so erweitert, daß er für jede *neue* Disjunktion im blockierten Knoten eine Entsprechung bei den *alten* Disjunktionen des blockierenden Knotens verlangt. Denn wenn es eine solche gibt, können die Disjunktionen im blockierten Knoten analog zu denen im blockierenden bearbeitet werden, ohne daß ein Clash auftritt. Gleichheit der Mengen ist nicht notwendig, denn bearbeitete Disjunktionen im blockierenden Knoten sind aus den oben genannten Gründen irrelevant für die Blockierung.

Diese Heuristiken sind in der Funktion *choose-next-node* () implementiert, die *choose-next-formula* () ersetzt. Sie durchsucht die Todo-Listen in der entsprechenden Reihenfolge, wählt aus der ersten nichtleeren Liste den Knoten mit der niedrigsten Nummer aus und gibt diesen zurück. Die Funktion *satisfy* (NODE) und ihre Subroutinen erhalten folglich einen Knoten anstelle einer Formel als Parameter.

Kapitel 7

Effizienz der Heuristiken

Um die Effizienz der verschiedenen Heuristiken zu testen und zu vergleichen, wurden Benchmark-Formeln mit unterschiedlichem Aufbau (zufällig oder systematisch erzeugt) und für unterschiedlich ausdrucksstarke Logiken (**K** bis $\mathcal{GF}$) verwendet. Die Verwendung von einfacheren Logiken zeigt dabei, ob SAGA gut *skaliert*, d. h. einfachere Formeln in einer Zeit löst, die vergleichbar ist mit der von Algorithmen, die auf diese Logiken spezialisiert sind (siehe Kapitel 8).

7.1 Verwendete Benchmark-Formeln

Es wurden für den Test vier Sätze von Benchmark-Formeln für verschieden ausdrucksstarke Logiken verwendet, die im folgenden beschrieben werden.

7.1.1 Logics Workbench

Die Benchmark-Formeln der *Logics Workbench (LWB)* sind in [BHS00] beschrieben. Für den Test von SAGA wurden die LWB-Formeln für die Logik **K** verwendet. Dieses System von Formeln besteht aus neun verschiedenen Sätzen von Formeln in gültigen und nicht gültigen Varianten und jeweils 21 Schwierigkeitsgraden, also insgesamt 378 Formeln. Die Leistung eines Programms wird gemessen in der Anzahl der Formeln, die es innerhalb des Zeitlimits von 100 Sekunden lösen kann. Da SAGA Erfüllbarkeit und nicht Gültigkeit einer Formel testet, wird jeweils überprüft, ob die Negation der entsprechenden Formel erfüllbar ist: wenn ein Tableau gefunden wird, war die ursprüngliche Formel nicht gültig.

Die systematische Struktur dieser Formeln hat mehrere Vorteile:

- Es sind auch sehr leichte Formeln enthalten, so daß auch bei ungünstigen Kombinationen von Heuristiken noch genug Formeln lösbar bleiben, um vergleichbare Ergebnisse zu erhalten.
- Die Formeln werden schrittweise schwieriger. Bei zufällig erzeugten Formelmengen ist es nicht klar, wie viel effizienter ein Algorithmus ist, der eine Formel mehr löst.
- Jede Formel desselben Satzes ist schwerer als die vorhergehende. Deshalb kann der Test der restlichen Formeln eines Satzes übersprungen werden, sobald eine nicht im Zeitlimit gelöst werden konnte.
- Da die richtige Antwort im voraus bekannt ist, kann auch die Korrektheit der Implementierung getestet werden.

Der offensichtliche Nachteil der LWB-Formeln ist, daß **K** wesentlich weniger ausdrucksstark ist als $\mathcal{GF}$, wodurch die Ergebnisse nicht unbedingt aussagekräftig für das Verhalten von SAGA sind. Die einfache Struktur der Logik **K** erlaubt es aber, den Test auf Blockierung zu deaktivieren (weil für **K**-Formeln die Quantorentiefe in jedem Knoten geringer ist als im Vaterknoten, siehe Abschnitt 4.1.2) und so den Aufwand, der durch den Blockierungstest entsteht, zu messen.

7.1.2 Tableaux Systems Comparison

Die Formeln der *Tableaux-2000 Non-Classical (Modal) Systems Comparison (TANCS)* [MD00, Mas99] sind unterteilt in mehrere Formelsätze für unterschiedlich ausdrucksstarke Logiken. Im Gegensatz zu den LWB-Formeln sind alle Formeln für eine Logik nach einem feststehenden Schema zufällig erzeugt. Die Leistungsfähigkeit eines Programms wird gemessen mit der Anzahl von Formeln, die innerhalb von 10 Minuten gelöst werden können.

QBF mit Inversen Für die Formeln des Satzes „QBF-INV-Kk-Cc-Vv-Dd“ wird zunächst eine Quantifizierte Boole'sche Formel (QBF) mit Quantorentiefe d+1, v Variablen pro Quantor und c Klauseln erzeugt. Jede dieser Klauseln enthält k Variablen, wobei die 1., 3. usw. existentiell und die 2., 4. usw. universell quantifiziert sind. Jede Variable ist mit einer Wahrscheinlichkeit von 50% negiert. Eine Formel für $d = 3, v = 2, c = 3, k = 4$ ist z. B.

$$\forall v_1 v_2 \exists v_3 v_4 \forall v_5 v_6 \exists v_7 v_8 (\neg v_3 \vee v_2 \vee v_8 \vee v_2) \wedge (v_4 \vee \neg v_6 \vee v_7 \vee \neg v_1) \wedge (v_4 \vee \neg v_2 \vee \neg v_7 \vee v_5)$$

Diese Formel kann auf verschiedene Weisen, von denen eine angelehnt ist an ein Verfahren von Schmidt-Schauss und Smolka [SS91], in die Logik $\mathbf{K}^-$ (bzw. nach $\mathcal{ALCI}$ [Spa93], $\mathcal{ALC}$ mit inversen Rollen) übersetzt werden, so daß die $\mathbf{K}^-$-Formel genau dann erfüllbar ist, wenn die QBF-Formel gültig ist.

Für den Test wurden die Formeln „p-qbf-inv-cnfSSS-K4-Cc-V4-D4“ mit $c \in \{10, 20, 30, 40, 50\}$ verwendet. Für jeden Wert von c gibt es 8 Formeln, also insgesamt 40.

PSAT mit Inversen Die Formeln des Satzes „PSAT-INV“ entstehen durch die Übersetzung einer Formel für „periodische“ Erfüllbarkeit (periodic satisfiability [Orl81]) in die Logik $\mathbf{K}^-$ mit globalen Axiomen. Es gibt verschiedene „Zeitpunkte“ und für jeden Zeitpunkt v Aussagenvariablen. Eine Formel besteht aus c Klauseln mit jeweils k Variablen, die mit einer Wahrscheinlichkeit von 50% negiert sind. Jedes Literal kann dabei eine aussagenlogisches Literal l sein (und sich so auf den gegenwärtigen Zeitpunkt beziehen) oder bis zu d (möglicherweise invertierte) $\Box$-Operatoren enthalten (und sich auf einen vergangenen oder zukünftigen Zeitpunkt beziehen). Das erste Literal bezieht sich immer auf die Gegenwart, das zweite auf einen anderen Zeitpunkt. Die restlichen werden zufällig ausgewählt. Ein Beispiel für $c = 2, k = 3, v = 2, d = 4$ ist

$$(v_1 \vee \Box\neg v_2 \vee \Box^{-1}v_1) \wedge (\neg v_2 \vee \Box^{-1}\Box v_2 \vee \Box\Box^{-1}\Box\Box\neg v_1)$$

Für den Test wurden die Formeln „p-psat-inv-cnf-K4-Cc-V4-D1“ mit $c \in \{20, 30, 40, 50\}$ verwendet. Es gibt ebenfalls 8 Formeln pro Parametersatz, also insgesamt 32.

7.1.3 $\mathcal{GF}$-Benchmark-Formeln

Keiner der bisher beschriebenen Benchmarks enthält n-stellige Relationen für $n > 2$ oder Gleichheitsassertionen für Variablen. Um SAGA auch mit Formeln aus der Logik zu testen, für die es entworfen worden ist, wurde deshalb ein Satz von $\mathcal{GF}$-Benchmark-Formeln (*GFB*) entwickelt, die allerdings nach einem einfacheren Schema als die bisher erwähnten Benchmarks erzeugt wurden. Die Entwicklung eines anspruchsvollen Benchmarks hätte den Rahmen dieser Arbeit gesprengt.

Eine GFB-Formel ist charakterisiert durch ihre Tiefe d, ihre Breite w und die Anzahl (und Stelligkeit) der in ihr vorkommenden Relationen a. Es gibt a verschiedene Relationen $R_i, i = 1, \ldots, a$; hierbei hat jedes R_i die

Stelligkeit i. Eine Formel der Breite w ist eine Konjunktion von w Formeln der Breite 1. Eine Formel von Breite 1 und Tiefe d mit aktuellen Variablen $\mathbf{y}$ ist

- eine Konjunktion $\varphi_1(\mathbf{y}) \wedge \varphi_2(\mathbf{y})$ oder
- eine Disjunktion $\varphi_1(\mathbf{y}) \vee \varphi_2(\mathbf{y})$ oder
- eine existentielle Formel $\exists \mathbf{x} G(\mathbf{y}, \mathbf{x}) \wedge \varphi_1(\mathbf{y}, \mathbf{x})$ mit neuen Variablen $\mathbf{x}$ (mit $|\mathbf{x}| \geq 1$) oder
- eine universelle Formel $\forall \mathbf{x} G(\mathbf{y}, \mathbf{x}) \rightarrow \varphi_1(\mathbf{y}, \mathbf{x})$.

Dabei ist das Guard $G(\mathbf{x}, \mathbf{y})$ entweder eine Relation $R_i(\mathbf{x}, \mathbf{y})$ oder eine Gleichheitsassertion $x = x$ (so daß die Formel ein globales Axiom ist). Die Formeln φ_i haben die Tiefe $d - 1$. Eine Formel der Tiefe 1 mit aktuellen Variablen $\mathbf{y}$ ist eine Relation $R_i(\mathbf{x})$ und $\mathbf{x} \subseteq \mathbf{y}$ oder eine Gleichheitsassertion $x_1 = x_2$ mit $\{x_1, x_2\} \subseteq \mathbf{y}$.

Zusätzlich enthält jede Formel für jede zwei- oder mehrstellige Relation globale Axiome, in denen für jede Konstante und jede Stelle einer Relation ein entsprechender Nachfolger gefordert wird. So ergeben sich für die zweistellige Relation R die Axiome

$$\forall x((x = x) \rightarrow \exists y((Rxy) \wedge \top))$$
$$\forall x((x = x) \rightarrow \exists y((Ryx) \wedge \top))$$

Das Erzeugen von Rollennachfolgern wird erzwungen, damit die $\forall$-Regel häufiger angewendet und so verhindert wird, daß universelle Formeln dadurch erfüllt sind, daß es keine Atome gibt, die eine Instanz des Guard sind.

Als Testformeln dienten Formeln mit den Parametern $w \in \{4, 8\}$, $d \in \{8, 16\}$, $r \in \{2, 4\}$. Für jede Kombination werden 8 Formeln erzeugt, also gibt es insgesamt 64 Formeln. Das Zeitlimit ist 100 Sekunden.

7.2 Durchführung der Tests

Um auch die Wechselwirkungen zwischen den einzelnen Heuristiken zu untersuchen, wurden für jeden Benchmark alle möglichen Kombinationen von Heuristiken getestet. Die Effizienz einer Heuristik A wurde dann bewertet, indem für jede Kombination der verbleibenden Heuristiken der Speedup ermittelt wurde, d. h. das Verhältnis von lösbaren Formeln mit aktiviertem A

zu deaktiviertem A.[1] Dann werden minimaler, maximaler und durchschnittlicher Speedup bestimmt.

Die getesteten Heuristiken sind:

1. Semantic Branching
2. Backjumping
3. Syntaktische Vereinfachung
4. die Verzweigungs-Heuristiken Maximize-Jump, MOMS und iMOMS
5. die Reihenfolge der Formeln: Bearbeitung von existentiellen Formeln vor oder nach Disjunktionen
6. Blockierung

Hierbei setzen die Heuristiken MOMS und iMOMS Semantic Branching voraus (weil sie für das häufigste Disjunkt φ die Alternativen φ und $\neg\varphi$ testen, siehe Abschnitt 6.2). Blockierung kann nur für die Formeln getestet werden, bei denen Terminierung aufgrund der Eigenschaften der Logik automatisch gegeben ist, also für LWB und QBF (siehe Abschnitt 4.1.2).

Alle Tests wurden auf demselben Computer mit der folgenden Ausstattung durchgeführt: Hardware: Pentium-III (733 MHz), 384 MB RAM, 512 MB Swap-Space. Software: Linux (Kernel 2.2), Allegro Common Lisp 6.0.

7.3 Resultate

In den folgenden Tabellen werden für den entsprechenden Benchmark und jede Heuristik maximaler, minimaler und durchschnittlicher Speedup angegeben. Falls Wechselwirkungen zwischen zwei Heuristiken zu beobachten sind, werden diese im Text erwähnt.

Der Speedup wird in Prozent angegeben; dieser Prozentsatz bezieht sich auf die Anzahl der Formeln, die gelöst werden konnten. Da die Formeln nicht notwendig linear schwerer werden, kann man diese Zahl nicht als Maß für die Zunahme der Leistungsfähigkeit auffassen, sondern nur als Indiz für eine mehr oder weniger starke Verbesserung.

[1] Wenn sich dabei ein Bruch ergibt, dessen Nenner 0 ist, wird der Nenner gleich 1 gesetzt. Damit ergibt sich für den Fall, daß durch eine Heuristik die Anzahl der lösbaren Formeln von 0 auf 5 gesteigert werden kann, ein Speedup von 500%. Auch wenn dieses Verfahren mathematisch nicht korrekt ist, gibt es doch einen besseren Eindruck von der Leistungsfähigkeit der Heuristik als das Ignorieren des entsprechenden Wertes.

Die verschiedenen Branching-Heuristiken Maximize-Jump, MOMS und iMOMS werden folgendermaßen miteinander verglichen: Da MOMS und iMOMS Semantic Branching voraussetzen, wird die Effizienz des Semantic Branching mit Maximize-Jump-Heuristik getestet. Der Speedup von MOMS und iMOMS wird dann jeweils im Verhältnis zu Maximize-Jump angegeben.

7.3.1 Logics Workbench

Tabelle 7.1 zeigt die Speedups der verschiedenen Heuristiken, die Abbildungen 7.1 und 7.2 die Zahl der lösbaren Formeln für den verschiedenen Kombinationen.

Heuristik	Max	Min	∅
Semantic Branching	104	47	73
Backjumping	77	12	43
Syntaktische Vereinfachung	23	-7	5
MOMS	-14	-40	-25
iMOMS	1	-10	-4
Existentielle Formeln früh	23	-3	5
Blocking	33	8	18

Tabelle 7.1: Heuristik-Vergleich für LWB

Semantic Branching und Backjumping erweisen sich als die besten Optimierungen. Bei Semantic Branching fällt zusätzlich auf, daß es die höchsten Speedups dann erreicht, wenn auch die ursprüngliche Leistung schon gut war. Syntaktische Vereinfachung dagegen ist nur bei aktiviertem MOMS spürbar effizienter (durchschnittlich 14%), während es sonst irrelevant ist (durchschnittlich 2%). Dieses deutet darauf hin, daß bei einer leistungsfähigen Kombination von Heuristiken eine nur syntaktisch komplizierte Formel den Algorithmus nicht merklich verlangsamt.

Wie bei GOST [Hla00] zeigt sich, daß Maximize-Jump und iMOMS ungefähr gleich effizient sind, MOMS aber deutlich schlechter ist. Dies liegt offensichtlich daran, daß zuerst die Alternative getestet wird, für die ein Scheitern wahrscheinlicher ist, denn auch bei deaktiviertem Backjumping ist die Maximize-Jump-Heuristik noch deutlich besser als MOMS, obwohl sie für diesen Fall der Auswahl einer zufälligen Formel entspricht (da es keinen Backjump gibt, der zu maximieren wäre).

Die Reihenfolge der Formeln (*don't-care*-Nichtdeterminismus) wirkt sich nur bei aktiviertem Backjumping aus, spielt aber auch dann eine unter-

geordnete Rolle. Überraschend ist dagegen, daß sich das Aktivieren des Blockierungstests positiv auswirkt. Er ist also so effizient, daß er weniger Zeit verbraucht als die Expansion der „blockierbaren“ Knoten.

In den folgenden Grafiken wird die wenig einflußreiche Heuristik „Syntaktische Vereinfachung“ nicht extra aufgeführt (sie ist für alle Werte aktiviert), um die Übersichtlichkeit zu erhöhen.

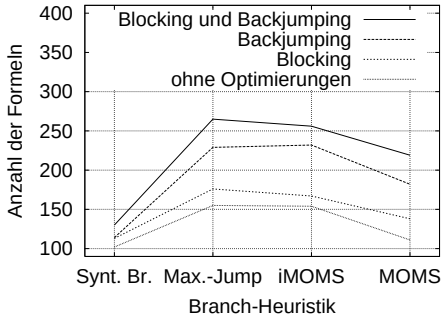

Abbildung 7.1: Heuristik-Vergleich LWB, Existentielle Formeln früh

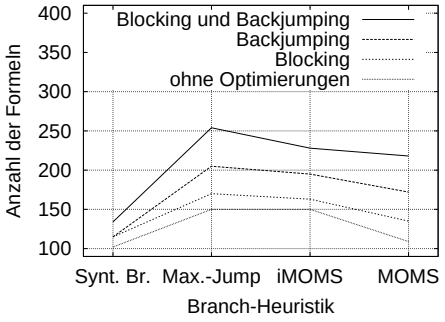

Abbildung 7.2: Heuristik-Vergleich LWB, Existentielle Formeln spät

7.3.2 QBF

Heuristik	Max	Min	∅
Semantic Branching	160	0	50
Backjumping	167	0	38
Syntaktische Vereinfachung	0	0	0
MOMS	-41	-100	-79
iMOMS	0	-75	-42
Existentielle Formeln früh	44	-25	3
Blocking	2300	233	780

Tabelle 7.2: Heuristik-Vergleich für QBF

Hier ist überraschenderweise Blockierung die entscheidende Optimierung: Ohne sie können höchstens 4 Formeln gelöst werden, mit ihr bis zu 26. Die Effizienz von Semantic Branching und Backjumping wird ebenfalls noch deutlicher als bei LWB, und wiederum erreicht Semantic Branching

die besten Speedups bei hohen Ausgangswerten.

Ebenfalls effizient ist Maximize-Jump, denn MOMS und iMOMS bleiben deutlich zurück. Die Struktur der Formeln scheint Syntaktische Vereinfachung zu verhindern: Sie hat überhaupt keinen Einfluß. Die Reihenfolge der Formeln hat wieder eine uneinheitliche Wirkung.

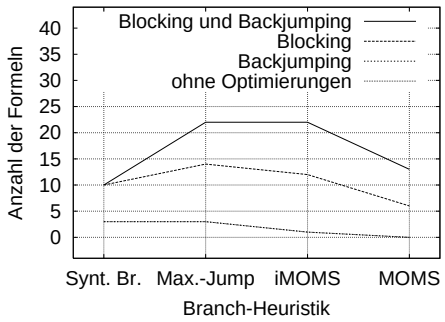

Abbildung 7.3: Heuristik-Vergleich für QBF; Existentielle Formeln früh

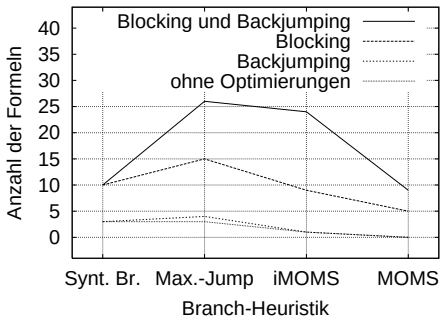

Abbildung 7.4: Heuristik-Vergleich für QBF; Existentielle Formeln spät

7.3.3 PSAT

Heuristik	Max	Min	∅
Semantic Branching	2400	50	700
Backjumping	700	0	125
Syntaktische Vereinfachung	80	-100	-22
MOMS	-33	-100	-75
iMOMS	167	0	69
Existentielle Formeln früh	0	-100	-46

Tabelle 7.3: Heuristik-Vergleich für PSAT

Der Einfluß der Heuristiken für Branching und die Reihenfolge der Formeln ist entgegengesetzt zu den bisherigen Resultaten: das Vorziehen der existentiellen Formeln wirkt sich deutlich nachteilig aus, und iMOMS ist besser als Maximize-Jump (allerdings bei guten Ausgangswerten nur geringfügig).

Semantic Branching erweist sich wieder als besonders wirksame Opti-

mierung, und auch Backjumping ist effizient.

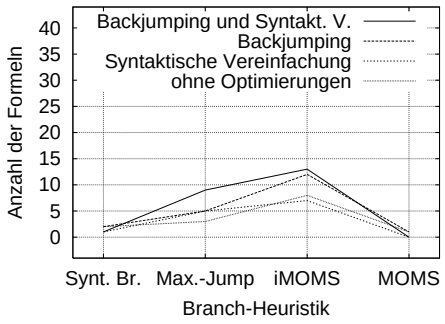

Abbildung 7.5: Heuristik-Vergleich für PSAT; Existentielle Formeln früh

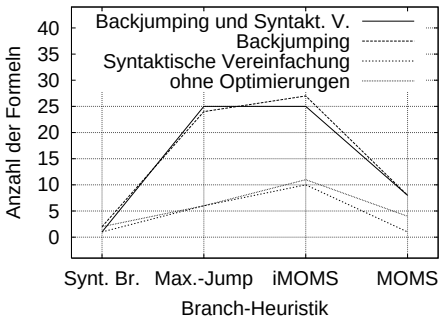

Abbildung 7.6: Heuristik-Vergleich für PSAT; Existentielle Formeln spät

7.3.4 GFB

Heuristik	Max	Min	∅
Semantic Branching	167	12	65
Backjumping	90	0	18
Syntaktische Vereinfachung	283	21	76
MOMS	-5	-62	-20
iMOMS	8	-38	-9
Existentielle Formeln früh	317	21	106

Tabelle 7.4: Heuristik-Vergleich für GFB

Im Gegensatz zu PSAT ist das Vorziehen der existentiellen Formeln die wichtigste Heuristik. Wenn sie aktiviert ist, haben auch die sonst dominierenden Optimierungen Semantic Branching und Backjumping nur noch geringen Einfluß (durchschnittlich 16% bzw. 2%). Die hohe Effizienz der Syntaktischen Vereinfachung liegt möglicherweise in der einfachen und zufälligen Struktur der Formeln begründet.

Semantic Branching, Backjumping und die Maximize-Jump-Heuristik wirken sich wiederum positiv aus, allerdings nicht so deutlich wie für die bisher betrachteten Formeln. Der Grund hierfür könnte sein, daß diese Techniken das Branching oder Backtracking optimieren, die GFB-Formeln aber weniger Disjunktionen enthalten als z. B. die TANCS-Formeln und

deshalb der Effizienzgewinn weniger deutlich ausfällt.

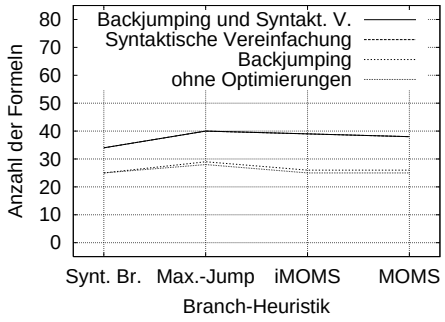

Abbildung 7.7: Heuristik-Vergleich für GFB, Existentielle Formeln früh

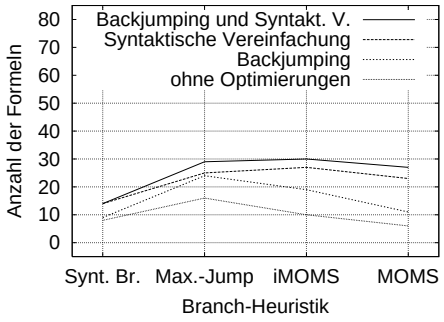

Abbildung 7.8: Heuristik-Vergleich für GFB, Existentielle Formeln spät

7.3.5 Zusammenfassung

Die Techniken Backjumping und Semantic Branching wirken sich durchgehend positiv aus und bringen teilweise einen erheblichen Geschwindigkeitszuwachs. Blockierung ist — selbst wenn der Blockierungstest derart aufwendig ist wie für $\mathcal{GF}$ — auch für solche Formeln nützlich, für die sie nicht zur Sicherstellung der Terminierung notwendig ist. In keinem Fall hat sie sich negativ ausgewirkt. Syntaktische Vereinfachung hat dagegen bei einer guten Kombination der anderen Heuristiken einen vernachlässigbaren Einfluß.

Bei den Verzweigungs-Heuristiken sind Maximize-Jump und iMOMS etwa gleich gut, wobei sich für die verschiedenen Benchmarks nur leichte Unterschiede zeigen. MOMS ist aber in allen Fällen deutlich schlechter. Das Vorziehen der existentiellen Formeln wirkt sich sehr uneinheitlich aus; es kann eine deutliche Verbesserung oder Verschlechterung bringen.

Kapitel 8

Vergleich mit anderen Systemen und Profiling

In diesem Kapitel wird die Effizienz von SAGA für die in Abschnitt 7.1 beschriebenen Benchmark-Formeln mit der anderer Systeme verglichen, um zu untersuchen, wie gut SAGA für diese Logiken skaliert. In Abschnitt 8.3 wird das Laufzeitverhalten der einzelnen Funktionen detailliert untersucht, um mögliche Schwächen in der Implementierung aufzudecken.

8.1 Vergleichssysteme

Zum Vergleich wurden die Systeme FaCT, SPASS, MSPASS und GOST herangezogen und die Benchmarks LWB, QBF, PSAT und GFB eingesetzt. Dabei verwenden FaCT und GOST Tableau-Algorithmen und auch weitgehend dieselben Optimierungen wie SAGA, während SPASS/MSPASS resolutionsbasiert arbeitet. Es konnte nicht jedes System mit jedem Benchmark getestet werden: Die GOST zugrundeliegende Logik $\mathcal{GF}1^-$ ist nicht ausdrucksstark genug, um die PSAT- oder GFB-Formeln ausdrücken zu können. Für MSPASS liegen nur Ergebnisse mit PSAT vor. SPASS ist neben SAGA das einzige System, das die GFB-Formeln verarbeiten kann, und es wurde auch nur mit diesen Formeln getestet.

8.1.1 FaCT

FaCT [Hor97] ist ein in Lisp implementiertes System, das mit Tableau-Algorithmen die ausdrucksstarken Beschreibungslogiken $\mathcal{SHF}$ ($\mathcal{ALC}$ erweitert um transitive Rollen, Rollenhierarchien und funktionale Rollen; in

[Hor97] unter dem Namen $\mathcal{ALCH}f_{R^+}$ beschrieben) und $\mathcal{SHIQ}$ ($\mathcal{ALC}$ mit transitiven Rollen, Rollenhierarchien, inversen Rollen und qualifizierenden Zahlenrestriktionen [HST99]) entscheidet. Dabei wurden die Resultate für LWB mit dem $\mathcal{SHF}$-Algorithmus erzeugt [HPS98], während die TANCS-Formeln wegen der in ihnen vorkommenden inversen Rollen den $\mathcal{SHIQ}$-Algorithmus erfordern [Hor00].

Die eingesetzten Optimierungen sind außer Syntaktischer Vereinfachung, Backjumping, Semantic Branching und BCP, die auch für SAGA verwendet werden, zusätzlich „GCI Absorption“, wodurch die Anzahl der Disjunktionen verringert wird, die durch „Global Concept Inclusion Axioms (GCIs)“ in jedem Knoten vorhanden sind, und „Caching“, das die Erfüllbarkeit oder Unerfüllbarkeit von Teilformeln zwischenspeichert und so vermeidet, für eine mehrfach vorkommende Teilformel auch mehrmals ein Tableau erzeugen zu müssen. Da beim Vorhandensein inverser Rollen eine Subformel aber an einer Stelle erfüllbar und an einer anderen unerfüllbar sein kann, wird Caching nur im $\mathcal{SHF}$- und nicht im $\mathcal{SHIQ}$-Algorithmus verwendet.

8.1.2 Gost

GOST [Hla00] ist SAGA ähnlich: Es ist ebenfalls in Lisp implementiert und enthält dieselben Optimierungen. Es wird aber die Tatsache ausgenutzt, daß für $\mathcal{GF}1^-$ eine Baum-Modell-Eigenschaft mit einem einfacheren Baum als einem k-Baum gilt [LST99]: jede Konstante kann in höchstens zwei benachbarten Knoten auftreten. Deshalb ist kein Propagieren von Formeln notwendig. Zudem gibt es in GOST keinen Blockierungstest, da $\mathcal{GF}1^-$ die Endliche-Baum-Modell-Eigenschaft hat und deswegen die Terminierung des Algorithmus auch ohne Blockierung gesichert ist.

Ein weiterer Unterschied zu SAGA liegt in der Art, wie bei einer Verzweigung die Sicherungskopie des Baums erstellt wird: In GOST werden nicht nur die geänderten Knoten gesichert, sondern es wird vor der Verzweigung der gesamte Baum kopiert. Dies führt zu einem deutlich erhöhten Speicherverbrauch und ist besonders für solche Formeln unnötig, die keine inversen Rollen enthalten, da es in diesem Fall reichen würde, beim Backtracking den bei der Bearbeitung der entsprechenden Formel erzeugten Teilbaum zu löschen. Andererseits werden durch dieses Vorgehen die notwendigen Verwaltungsinformationen in den Datenstrukturen reduziert (z. B. sind die Datenstruktur „Verzweigungspunkt“ und die Felder CREATED-BY und NEWEST-BID der Datenstruktur „Knoten“ nicht notwendig), und beim Backjumping ist das Ersetzen des Baums durch seine Kopie sehr schnell. Welcher Effekt überwiegt, läßt sich aus dem Vergleich mit SAGA ableiten.

8.1.3 Spass und MSpass

SPASS [Wei99a, WAB+99, Wei99b] ist ein resolutionsbasierter Theorembeweiser für Prädikatenlogik erster Stufe mit Gleichheit. Eine Formel wird zunächst in eine Klauselmenge übersetzt, auf die dann Resolution angewendet wird. Da Erfüllbarkeit für FO-Formeln unentscheidbar ist, terminiert SPASS nicht notwendig. MSPASS [Sch99] erweitert SPASS um die Syntax von Modal- und Beschreibungslogiken. Eine entsprechende Formel wird hier zuerst in eine prädikatenlogische Formel übersetzt und anschließend mit SPASS auf Erfüllbarkeit getestet.

SPASS und MSPASS sind in ANSI C implementiert. Sie enthalten, ebenso wie die anderen beschriebenen Systeme, zahlreiche Optimierungen, darunter „Splitting“ und „Branch Condensing“, wobei das letztere Backjumping ähnelt [HS00]. Damit war MSPASS das effizienteste System bei TANCS-2000 [HS00, MD00].

8.2 Resultate

In Abbildung 8.1 wird, anders als in Kapitel 7, nicht nur die Summe der lösbaren Formeln angegeben, sondern werden die Ergebnisse nach den einzelnen Formelsätzen aufgeschlüsselt.

Da nicht alle Tests für diese Arbeit neu durchgeführt werden konnten, mußten einige Ergebnisse aus anderen Arbeiten übernommen werden. Deshalb gibt es zum Teil unterschiedliche Plattformen:

- SAGA: siehe Abschnitt 7.2.
- FaCT: **LWB**: Hardware: Sun Ultra 1 (147 MHz), 32 MB. Software: Solaris, Allegro CL 4.3. **TANCS**: Hardware: 450 MHz Pentium-III, 128 MB, und 433 MHz Celeron, 256 MB. Software: Linux, Allegro CL 5.0.
- GOST: **LWB**: Hardware: Pentium-III-450, 256 MB. Software: Linux, Allegro CL 5.0. **TANCS**: siehe SAGA.
- MSPASS: Hardware: Cluster von Pentium-II-300 mit 256 MB. Software: Solaris 2.6.
- SPASS: siehe SAGA.

8.2.1 LWB

Die Effizienz von SAGA ist weitgehend identisch mit der von FaCT. Der Vergleich mit GOST zeigt wiederum, daß der Blockierungstest und die aufwendigere Backup-Strategie den Algorithmus nicht verlangsamen, sondern beschleunigen.

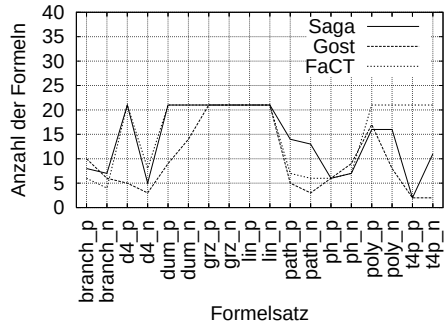

Abbildung 8.1: System-Vergleich LWB

8.2.2 QBF

Um eine größere Menge unterschiedlicher Formeln zu testen, wurden zusätzlich zu den Formeln „p-qbf-inv-cnfSSS-K4-Cx-V4-D4“ (siehe Abschnitt 7.1.2) die Sätze mit den Parametern „V4-D6“ und „V8-D4“ verwendet.

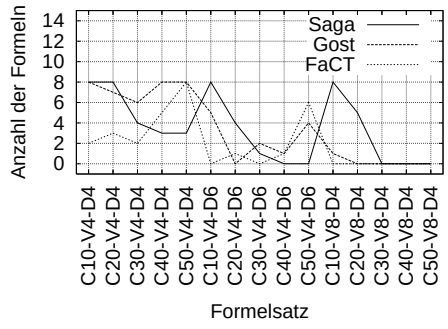

Abbildung 8.2: System-Vergleich QBF

Der Verlauf der Kurve für SAGA ist hier umgekehrt zu der für FaCT: Die

(größtenteils erfüllbaren) Formeln mit wenigen Klauseln sind für SAGA leicht zu lösen, die (größtenteils unerfüllbaren) mit vielen Klauseln können häufig nicht innerhalb des Zeitlimits entschieden werden. Auch GOST ist für diese Formeln effizienter als SAGA. Eine Erklärung hierfür ist, daß für unerfüllbare Formeln eine maximale Anzahl von Backtracking-Schritten notwendig wird und das Rückgängigmachen der Änderungen für SAGA zeitaufwendiger ist. Allerdings zeigt sich für die komplexeren Formeln die Ineffizienz der naiven Backup-Strategie von GOST: Die meisten Formeln führen schon nach kurzer Zeit zu einem vollen Speicher.

Diese Ergebnisse werden in Tabelle 8.1 noch deutlicher sichtbar. Es wird für jeden Formelsatz angegeben, wie viele Formeln als erfüllbar (E) oder unerfüllbar (U) erkannt wurden, und wie viele zu einem Timeout (T) oder Speicherüberlauf (M) führten. Hier erkennt man, daß die schnelle, aber viel Platz verbrauchende Backup-Strategie von GOST für unentscheidbare einfache Formeln leistungsfähiger ist als die von SAGA, aber bei komplexeren Formeln viel häufiger zuviel Speicher verbraucht.

	SAGA				GOST				FaCT			
Formel	E	U	T	M	E	U	T	M	E	U	T	M
C10-V4-D4	8				8				2			6
C20-V4-D4	5	3			5	2	1		1	2		5
C30-V4-D4	1	3	4		3	5			1	1		6
C40-V4-D4		3	5			8				5		3
C50-V4-D4		3	5			8				8		
C10-V4-D6	8				5			3				8
C20-V4-D6	3	1	4				1	7		1		7
C30-V4-D6	1		7			2	2	4		1		7
C40-V4-D6			8			1	3	4		1		7
C50-V4-D6			8			4	4			6		2
C10-V8-D4	8				1		1	6				8
C20-V8-D4	5		3					8				8
C30-V8-D4			8				2	6				8
C40-V8-D4			8				4	4				8
C50-V8-D4			8				6	2				8

Tabelle 8.1: System-Vergleich QBF

8.2.3 PSAT

Zum Vergleich werden die Formelsätze „p-psat-inv-cnf-K4-Cx-Vy-D1“ verwendet. Da im Test von MSpass ein anderer Timeout verwendet wird als in dem von FaCT (120 statt 10 Minuten), werden die Saga-Ergebnisse getrennt mit denen von FaCT und MSpass verglichen.

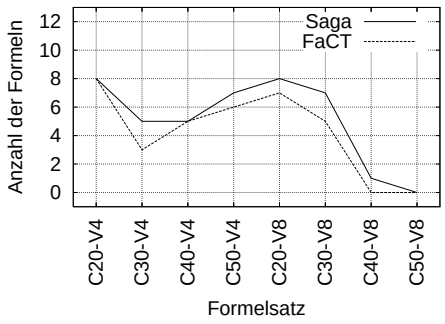

Abbildung 8.3: System-Vergleich PSAT, Timeout 10 min

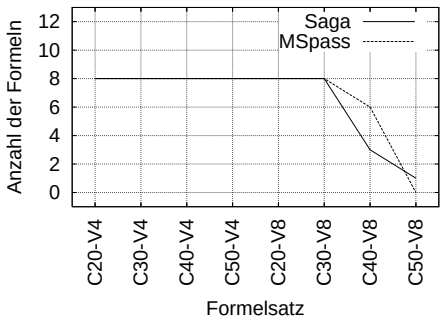

Abbildung 8.4: System-Vergleich PSAT, Timeout 120 min

Mit dem entsprechenden Timeout ist die Effizienz von Saga annähernd identisch mit der des jeweiligen anderen Systems. Allerdings ist im Vergleich mit MSpass zwar die Anzahl der lösbaren Formeln sehr ähnlich, die durchschnittlich benötigte Zeit bei Saga aber höher. Hinzu kommt, daß fast alle lösbaren Formeln erfüllbar waren und daß schon die vorherigen Tests gezeigt haben, daß Saga im Vergleich mit den anderen Systemen bei erfüllbaren Formeln besser abschneidet als bei unerfüllbaren.

	Saga		MSpass		FaCT	
Formel	#	t	#	t	#	t
C20-V4	8	28	8	1	8	2
C30-V4	8	238	8	4	3	13
C40-V4	8	230	8	3	5	64
C50-V4	8	413	8	2	6	2
C20-V8	8	2	8	1	7	< 1
C30-V8	8	136	8	57	5	14
C40-V8	3	2483	6	1995	0	–
C50-V8	1	715	0	–	0	–

Tabelle 8.2: System-Vergleich PSAT

In Tabelle 8.2 wird für jeden Formelsatz die Anzahl der lösbaren Formeln (#) und der Median der benötigten Zeit (t, in Sekunden) angegeben. Die Werte für SAGA sind hier die mit dem Timeout 7200 Sekunden gemessenen. Insgesamt ist SAGA zwischen FaCT und MSPASS einzuordnen.

8.2.4 GFB

Ziel dieses Tests ist, herauszufinden, ob die Einschränkung der Ausdrucksstärke auf $\mathcal{GF}$ (gegenüber FO) auch einen Effizienzgewinn bringt. Es wurden die in Abschnitt 7.1.3 beschriebenen Formeln eingesetzt, der Timeout war ebenfalls 100 Sekunden.

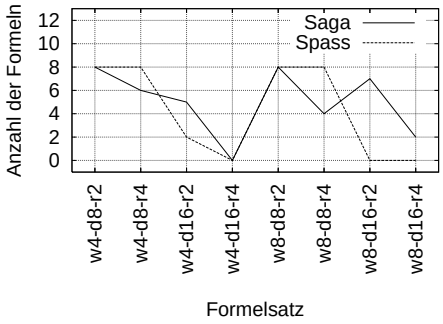

Abbildung 8.5: System-Vergleich GFB

In der Tat zeigen sich deutliche Unterschiede im Laufzeitverhalten: Während SAGA auch zahlreiche Formeln mit großer Tiefe entscheiden kann, aber bei einer höheren Anzahl von Relationen langsamer wird, kann SPASS alle Formeln mit Tiefe 8 schnell lösen (in der Regel werden weniger als 2 Sekunden benötigt), aber fast keine Formeln der Tiefe 16. Die naheliegende Vermutung, daß die Stärken eines Tableau-Algorithmus eher bei erfüllbaren und der eines Resolutions-Algorithmus eher bei unerfüllbaren Formeln liegt, bestätigte sich nicht: Es gibt sowohl erfüllbare Formeln, die von SPASS innerhalb einer Sekunde und von SAGA nicht innerhalb des Timeout gelöst werden konnten (w8-d8-r4), wie auch den umgekehrten Fall (w4-d16-r2).

Tabelle 8.3 zeigt für jeden Formelsatz, wie viele erfüllbare (E) und unerfüllbare (U) Formeln in welcher Zeit (t; angegeben wird der Median) entschieden werden konnten.

	SAGA				SPASS			
	E		U		E		U	
Formel	#	t	#	t	#	t	#	t
w4-d8-r2	1	7	7	< 1	1	< 1	7	< 1
w4-d8-r4	4	16	2	8	6	< 1	2	< 1
w4-d16-r2	0	–	5	1	0	–	2	16
w4-d16-r4	0	–	0	–	0	–	0	–
w8-d8-r2	0	–	8	< 1	0	–	8	< 1
w8-d8-r4	0	–	4	< 1	3	1	5	< 1
w8-d16-r2	0	–	7	3	0	–	0	–
w8-d16-r4	0	–	2	58	0	–	0	–

Tabelle 8.3: System-Vergleich GFB

8.3 Profiling

Um einen Eindruck davon zu bekommen, wie aufwendig die verschiedenen Operationen sind, wurde für jede einzelne Funktion gemessen, welcher Anteil der insgesamt verbrauchten Zeit ihr zukommt. Hierzu wurde ein kompletter Durchlauf des QBF-Benchmarks (mit den für diese Formeln optimalen Parametern) analysiert. Dieser Benchmark wurde ausgewählt, weil hier für unerfüllbare Formeln die Leistungsfähigkeit zum Teil deutlich hinter der anderer Systeme zurückblieb (siehe Abschnitt 8.2.2) und deshalb die Schwächen von SAGA am deutlichsten werden. Die Resultate sind in Tabelle 8.4 angegeben.

In der Implementierung von SAGA ist die Funktionalität von *choose-alternative* in *satisfy-or* enthalten und taucht deswegen nicht als eigenständige Funktion auf. Da die QBF-Formeln keine Gleichheitsassertionen enthalten, hat *satisfy-eq* keinen Anteil an der Gesamtlaufzeit.

Diese Messung zeigt sehr deutlich, warum SAGA gerade für unerfüllbare Formeln langsamer ist als andere Systeme: Die Hilfsfunktion *propagate-to*, die nur die seltenen Fälle abfangen soll, in denen eine durch Propagieren in einen Knoten gelangte Formel beim Wiederherstellen einer Kopie verlorengeht, verbraucht fast zwei Drittel der gesamten Zeit. Demgegenüber verbrauchen Funktionen, bei denen man einen hohen Anteil an der Gesamtlaufzeit erwarten könnte, nur wenig Zeit: der Blockierungstest nur 2,3%, und das Backjumping, nach Abzug der auf erneutes Propagieren entfallenden Zeit, nur 0,2%. Der für das Verzweigen und Erstellen der Knoten-Kopien benötigte Anteil ist kaum meßbar.

Funktion	Anteil
construct-tableau	100,0
propagate	77,7
add-formula	70,8
backjump	65,5
restore-tree	65,4
propagate-to	65,3
normalize-variables	55,5
substitute	53,2
satisfy	30,6
satisfy-all	20,8
constants	20,4
encode	10,3
satisfy-ex	7,4
contains	3,0
blocked	2,3
equivalent	2,1
identify	1,2
satisfy-or	1,1
match-const	1,0
satisfy-and	0,8
bcp	0,5
choose-next-formula	0,4
make-binding	0,3
simplify	0,3
decode	0,2
get-node	0,2
satisfy-atom	0,1
match-var	0,1
branch	<0,1
copy	<0,1
block	<0,1
get-bp	<0,1
unblock	<0,1
satisfy-eq	0,0

Tabelle 8.4: Profiling

Die Ineffizienz von *propagate-to* könnte vermieden werden, indem eine Formel beim Propagieren nicht nur dem Knoten selbst, sondern auch seinen Kopien hinzugefügt wird. Dafür müßte in jedem Knoten über die bestehenden Kopien Buch geführt werden. Dieses ist aber mit Sicherheit effizienter als das gegenwärtige Vorgehen mit der Funktion *propagate-to*.

Die Funktion *normalize-vars* verbraucht ebenfalls einen großen Anteil der Gesamtzeit, allerdings gehen etwa 70% dieser Zeit auf das Konto von Aufrufen, die durch *propagate-to* verursacht wurden, so daß in einer verbesserten Version ein geringerer Anteil zu erwarten ist. Ähnliches gilt für *substitute*.

Das Codieren einer Formel mit Hilfe der Hash-Tabelle ist erwartungsgemäß deutlich langsamer als das Decodieren mit einem Array (10,3% gegenüber 0,2%). Für die gewonnene Zeit- und Platz-Ersparnis erscheint dieser Anteil aber vertretbar.

Von den Funktionen für die Bearbeitung der verschiedenen Arten von Formeln sind *satisfy-all* und *satisfy-ex* am stärksten vertreten, was nicht verwunderlich ist, da hier ein Matching durchgeführt werden muß.

Kapitel 9

Fazit

In dieser Arbeit wurden Implementierung und Optimierung eines Tableau-Algorithmus für das Bewachte Fragment beschrieben. Grundlage hierfür waren die Definition und Untersuchung von $\mathcal{GF}$ in [AvBN98, Grä99b] einerseits und der in [HT01] entwickelte Tableau-Algorithmus andererseits.

Es hat sich gezeigt, daß die für Beschreibungs- und Modallogiken eingesetzten Optimierungen, insbesondere Semantic Branching und Backjumping, auch für $\mathcal{GF}$ hocheffizient sind und trotz der theoretisch hohen Komplexität zu praktischer Entscheidbarkeit führen. Syntaktische Vereinfachung hat aber, besonders bei einer leistungsfähigen Kombination der anderen Optimierungen, nur geringen Einfluß. Auch die Auswirkungen von verschiedenen Heuristiken für nichtdeterministische Entscheidungen waren uneinheitlich; es läßt sich nicht allgemein eine optimale Kombination geben. Erkennbar ist allerdings die Tendenz, daß es sinnvoller ist, zuerst einen Baum mit weniger Constraints zu untersuchen.

Eine unerwartete Beobachtung ist, daß sich das Aktivieren des Blockierungstests auch für solche Formeln positiv auswirkt, für die er wegen der zugrundeliegenden Logik nicht notwendig wäre. Die eingesetzten Tests für Nicht-Blockierung scheinen ihre Aufgabe zu erfüllen und die Nicht-Äquivalenz zweier Knoten ausreichend schnell zu erkennen.

Mit der oben beschriebenen Kombination von Optimierungen ist die Effizienz von Saga zwischen der von FaCT und Spass/MSpass einzuordnen. Dies ist überraschend, da die FaCT zugrundeliegend Logik $\mathcal{SHIQ}$ in ExpTime enthalten ist [HST99] und damit deutlich weniger komplex ist als $\mathcal{GF}$. Für unerfüllbare Formeln fällt der Vergleich aber ungünstiger aus als für erfüllbare. Die Begründung hierfür liegt im eingesetzten Verfahren zur Wiederherstellung des Baums beim Backjumping, das in der gegenwärtigen

Form mehr Zeit verbraucht als nötig.

Um die Effizienz, insbesondere für unerfüllbare Formeln, zu erhöhen, sollte das Vorgehen zur Wiederherstellung des Baums nach dem Entdecken eines Widerspruchs verbessert werden. Ein möglicher Weg hierzu ist in Abschnitt 8.3 skizziert.

Weiterhin ist eine Erweiterung auf das Clique-Bewachte Fragment naheliegend, wie sie auch schon in [HT01] beschrieben ist. Hierfür wären die Tests für die Anwendbarkeit von existentiellen und universellen Formeln zu modifizieren. Beide Modifikationen sind ohne grundlegende Änderungen an der Struktur von SAGA möglich, hätten aber den Rahmen dieser Arbeit gesprengt.

Literaturverzeichnis

[AvBN98] Andréka, H., J. van Benthem und I. Németi: *Modal Languages and Bounded Fragments of Predicate Logic.* Journal of Philosophical Logic, 27(3):217–274, 1998.

[Baa99] Baader, F.: *Logic-Based Knowledge Representation.* In: Wooldridge, M. J. und M. Veloso (Herausgeber): *Artificial Intelligence Today, Recent Trends and Developments*, Nummer 1600 in *Lecture Notes in Computer Science*, Seiten 13–41. Springer-Verlag, 1999.

[BdRV01] Blackburn, P., M. de Rijke und Y. Venema: *Modal Logic.* Cambridge University Press, 2001.

[BHS00] Balsiger, P., A. Heuerding und S. Schwendimann: *A Benchmark Method for the Propositional Modal Logics K, KT, S4.* Journal of Automated Reasoning, 24(3):297–317, April 2000.

[BS01] Baader, F. und U. Sattler: *An Overview of Tableau Algorithms for Description Logics.* Studia Logica, 2001.

[DFvH+00] Decker, S., D. Fensel, F. van Harmelen, I. Horrocks, S. Melnik, M. Klein und J. Broekstra: *Knowledge Representation on the Web.* In: *Proc. of the 2000 Int. Workshop on Description Logics (DL2000)*, Seiten 89–98, 2000.

[dN98] Nivelle, H. de: *A Resolution Decision Procedure for the Guarded Fragment.* In: Kirchner, C. und H. Kirchner (Herausgeber): *Proceedings of the 15th International Conference on Automated Deduction (CADE-98)*, Band 1421 der Reihe *Lecture Notes In Artificial Intelligence*, Seiten 191–204, Berlin, 1998. Springer-Verlag.

[dN01] NIVELLE, H. DE: *Translation from S4 into the guarded fragment and the 2-variable fragment*, 2001. `http://www.mpi-sb.mpg.de/~nivelle/slides/s4_slides.ps`.

[FL79] FISCHER, M. J. und R. E. LADNER: *Propositional Dynamic Logic of Regular Programs.* Journal of Computer and System Science, 18:194–211, 1979.

[Fre95] FREEMAN, J. W.: *Improvements to propositional satisfiability search algorithms.* Doktorarbeit, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA, 1995.

[GKV97] GRÄDEL, E., P. KOLAITIS und M. VARDI: *On the Decision Problem for Two-Variable First-Order Logic.* Bulletin of Symbolic Logic, 3:53–69, 1997.

[Grä99a] GRÄDEL, E.: *Decision procedures for guarded logics.* In: *Automated Deduction - CADE16. Proceedings of 16th International Conference on Automated Deduction, Trento, 1999*, Band 1632 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[Grä99b] GRÄDEL, E.: *On the restraining power of guards.* Journal of Symbolic Logic, 64:1719–1742, 1999.

[GW99] GRÄDEL, E. und I. WALUKIEWICZ: *Guarded Fixed Point Logic.* In: *Proceedings of 14th IEEE Symposium on Logic in Computer Science LICS '99, Trento*, Seiten 45–54, 1999.

[Hla00] HLADIK, J.: *Implementing the n-ary Description Logic GF1-.* In: *Proceedings of the International Workshop in Description Logics 2000 (DL2000)*, Aachen, 2000.

[HMT85] HENKIN, L., J. D. MONK und A. TARSKI: *Cylindric Algebras, Part II.* North-Holland, Amsterdam, 1985.

[HO01] HODKINSON, I. und M. OTTO: *Finite Conformal Hypergraph Covers, with Two Applications.* 2001.

[Hod00] HODKINSON, I.: *Loosely guarded fragment of first-order logic has the finite model property.* Studia Logica, 2000.

[Hor97] Horrocks, I.: *Optimising Tableaux Decision Procedures for Description Logics.* Doktorarbeit, University of Manchester, 1997.

[Hor00] Horrocks, I.: *Benchmark Analysis with FaCT.* In: *Proc. of the 4th Int. Conf. on Analytic Tableaux and Related Methods (TABLEAUX 2000)*, Band 1847 der Reihe *Lecture Notes In Artificial Intelligence*, Seiten 62–66. Springer-Verlag, 2000.

[HPS98] Horrocks, I. und P. F. Patel-Schneider: *FaCT and DLP.* In: Swart, H. de (Herausgeber): *Automated Reasoning with Analytic Tableaux and Related Methods*, Nummer 1397 in *Lecture Notes in Artificial Intelligence*, Seiten 27–30, Oisterwijk, Netherlands, Mai 1998. Springer-Verlag.

[HS99] Horrocks, I. und U. Sattler: *A Description Logic with Transitive and Inverse Roles and Role Hierarchies.* Journal of Logic and Computation, 1999.

[HS00] Hustadt, U. und R. A. Schmidt: *MSPASS: Modal Reasoning by Translation and First-Order Resolution.* In: Dyckhoff, R. (Herausgeber): *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000)*, Band 1847 der Reihe *Lecture Notes in Artificial Intelligence*, Seiten 67–71. Springer-Verlag, 2000.

[HST99] Horrocks, I., U. Sattler und S. Tobies: *Practical Reasoning for Expressive Description Logics.* In: Ganzinger, H., D. McAllester und A. Voronkov (Herausgeber): *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, Nummer 1705 in *Lecture Notes In Artificial Intelligence*, Seiten 161–180. Springer-Verlag, 1999.

[HST00] Horrocks, I., U. Sattler und S. Tobies: *Practical Reasoning for Very Expressive Description Logics.* Logic Journal of the IGPL, 8(3):239–264, Mai 2000.

[HT01] Hirsch, C. und S. Tobies: *A Tableau Algorithm for the Clique Guarded Fragment.* In: Wolter, F., H. Wansing, M. de Rijke und M. Zakharyaschev (Herausgeber): *Advances in Modal Logics Volume 3*, Stanford, 2001. CSLI Publications.

[Lad77] LADNER, R. E.: *The computational complexity of provability in systems of modal propositional logic.* SIAM Journal of Computing, 6(3):467–480, 1977.

[LST99] LUTZ, C., U. SATTLER und S. TOBIES: *A Suggestion for an n-ary Description Logic.* In: LAMBRIX, P., A. BORGIDA, M. LENZERINI, R. MÖLLER und P. PATEL-SCHNEIDER (Herausgeber): *Proceedings of the International Workshop on Description Logics*, Nummer 22 in *CEUR-WS*, Seiten 81–85, Linkoeping, 1999. Linköping University.

[Mas99] MASSACCI, F.: *Design and Results of the Tableaux-99 Non-Classical (Modal) Systems Comparison.* In: MURRAY, N. (Herausgeber): *Proceedings of the Third International Conference on Analytic Tableaux and Related Methods (TABLEAUX'99)*, Band 1617 der Reihe *Lecture Notes in Artificial Intelligence*, Seiten 14–18. Springer-Verlag, 1999.

[MD00] MASSACCI, F. und F. M. DONINI: *Design and Results of TANCS-00.* In: DYCKHOFF, R. (Herausgeber): *Proceedings of the Fourth International Conference on Analytic Tableaux and Related Methods (TABLEAUX 2000)*, Band 1847 der Reihe *Lecture Notes in Artificial Intelligence*, Seiten 52–56. Springer-Verlag, 2000.

[Mor75] MORTIMER, M.: *On Languages with two variables.* Zeitschr. f. math. Logik u. Grundlagen d. Math., 21:135–140, 1975.

[OKvLN95] OWSNICKI-KLEWE, B., K. VON LUCK und B. NEBEL: *Einführung in die Künstliche Intelligenz*, Kapitel 1.1, Seiten 15–58. Addison-Wesley, 2. Auflage, 1995.

[Orl81] ORLIN, J.: *The complexity of dynamic languages and dynamic optimization problems.* In: *Transactions of the 13th Annual ACM Symposium on The Theory of Computing*, Seiten 218–227. ACM Press, New York, Mai 1981.

[Pra79] PRATT, V. R.: *Models of Program Logics.* In: *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, 1979.

[Sat98] SATTLER, U.: *Terminological knowledge representation systems in a process engineering application.* Doktorarbeit, LuFG Theoretical Computer Science, RWTH-Aachen, 1998.

[Sch91] SCHILD, K.: *A Correspondence Theory for Terminological Logics: Preliminary Report.* In: *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI-91)*, Seiten 466–471, Sydney, 1991.

[Sch99] SCHMIDT, R. A.: *MSpass*, 1999. `http://www.cs.man.ac.uk/~schmidt/mspass/`.

[Spa93] SPAAN, E.: *The Complexity of Propositional Tense Logics.* In: RIJKE, M. DE (Herausgeber): *Diamonds and Defaults*, Seiten 287–307. Kluwer Academic Publishers, 1993.

[SS91] SCHMIDT-SCHAUSS, M. und G. SMOLKA: *Attributive Concept Descriptions with Complements.* Artificial Intelligence, 48(1):1–26, 1991.

[vB97] BENTHEM, J. VAN: *Dynamic bits and pieces.* ILLC research report LP-97-01, Institute for Logic, Language and Computation, University of Amsterdam, 1997.

[WAB+99] WEIDENBACH, C., B. AFSHORDEL, U. BRAHM, C. COHRS, T. ENGEL, E. KEEN, C. THEOBALT und D. TOPIĆ: *System Description: SPASS Version 1.0.0.* In: GANZINGER, H. (Herausgeber): *Proceedings of the 16th International Conference on Automated Deduction (CADE-16-99)*, Band 1632 der Reihe *Lecture Notes In Artificial Intelligence*, Seiten 378–382, Berlin, 1999. Springer-Verlag.

[Wei99a] WEIDENBACH, C.: *Handbook of Automated Reasoning*, Kapitel 27. Elsevier, 1999.

[Wei99b] WEIDENBACH, C.: *Spass*, 1999. `http://spass.mpi-sb.mpg.de`.