Projektgruppe Fuzzy^2

Repräsentation und Verarbeitung
unscharfen und unsicheren Wissens

Endbericht

Matthias Arentz
Alexander Daxenberger
André Grosskopf
Hanns Haustein
Lars Heyden
Sören Kerner
Matthias Langer
Iris Luhle
Gregor Manthey
Michael Scheibel
Thomas Umpfenbach

3.3.2004

Inhaltsverzeichnis

Kapitel 1

Einleitung

Es gibt viele Gelegenheiten, Vagheit (Unbestimmtheit, Verschwommenheit) und Unsicherheit in der Wissensrepräsentation explizit zu berücksichtigen.

Erstens ist es für die Wissensrepräsentation unvermeidlich, Begriffe zu bilden und reale Objekte in konzeptuell unterscheidbare Klassen einzuteilen. Trotzdem ist zu erwarten, daß ein reales Objekt niemals eindeutig einem Begriff oder einer Klasse zuzuordnen ist, sondern verschiedenen Begriffen bzw. Klassen jeweils zu einem bestimmten Grad.

Zweitens sind häufig bestimmte Daten oder Zusammenhänge, die in einer erstellten Wissensbasis eine Rolle spielen, in einer konkreten realen Situation nicht oder nur ungenau bekannt und es ist notwendig, die Zuverlässigkeit der gezogenen Schlüsse zu bewerten.

Zum dritten sind häufig repräsentierte Wissenszusammenhänge nicht absolut sicher, sondern stellen Daumenregeln, Hörensagen oder Vermutungen dar. Auch kann bei einer Wissensakquisition eine Quelle als nicht absolut zuverlässig bekannt sein, ohne daß auf ihren Beitrag verzichtet werden kann.

Eine vierte Unsicherheitsquelle liegt darin, daß präzise Informationen häufig mit Kosten verbunden sind (in Realzeitsystemen beziehen diese sich häufig auf die Zeit, die für exakte Berechnungen notwendig wäre), so daß auf exakte Daten zuzgunsten von Schätzungen oder potentiell veralteten Daten verzichtet wird.

Mittels markierter logischer Formeln einer mehrsortigen Logik kann ein Formalismus dargestellt werden, der es ermöglicht, verschiedene Aspekte von Vagheit und Unsicherheit zu kombinieren.

Die Projektgruppe 429

Fuzzy^2— Repräsentation und Vearbeitung unscharfen und unsicheren Wissens

hat auf der Basis theoretischer Untersuchungen (u.a. aus dem ersten PG-Semester) eine praktische Umsetzung eines Wissensrepräsentationssystems

erstellt, in dem Unsicherheit und Vagheit miteinander verknüpft werden. Dabei bekommt jede Formel (Aussage) ein sogenanntes Label. Dieses wird in einem Koordinatensystem veranschaulicht, in dem die Abszisse die Vagheit und die Ordinate die Unsicherheit repräsentiert.
Das System bedient sich bestimmter Werkzeuge. Zu nennen sind da das Anwenden von Axiomen auf eine Wissensbasis, Modus Ponens sowie Forward-Chaining.
Auf diese Weise ist es möglich, herauszufinden, ob sich eine Aussage aus einer Wissensbasis heraus ableiten lassen kann und damit „versteckte“ Informationen, d.h. Informationen, die nicht explizit notiert sind, gewinnen zu können. Da die gewonnenen Aussagen natürlich ebenfalls über ein Label verfügen, die aus den „Quellaussagen“ mittels geeigneter Verfahren berechnet werden, weiß man auch, mit welcher Sicherheit diese zutreffen.

Damit ist es dann automatisiert möglich, aus einer Wissensbasis heraus, Situationen zu bewerten und Entscheidungen treffen zu können.

Kapitel 2

Zusammenfassung des Zwischenberichts

Der Zwischenbericht fasst die in der PG erarbeiteten Ergebnisse und Zwischenschritte der Arbeit des ersten Semesters zusammen. Ziel der Projektarbeit war es ein Werkzeug zur Repräsentation und Verarbeitung von unscharfem und vagen Wissens zu erstellen, wobei der theoretische Hintergrund dabei im Wesentlichen auf der Diplomarbeit von Stefan Lehmke fußte. Gegen Ende des ersten Semesters wurde das noch zu entwickelnde Werkzeug umfassend spezifiziert sowie eine aussagenlogische Beschreibungssprache mit einfacher Wissensverarbeitung implementiert. Ein abschließendes Beispiel für eine prädikatenlogische Wissensrepräsentation gibt schließlich Ausblicke auf die geplante prädikatenlogische Erweiterung der Beschreibungssprache. Der Zwischenbericht dokumentiert den Weg zu diesen Resultaten. Er gliedert sich in sechs Unterkapitel (die Einleitung ausgenommen).

2.1 Abstracts der Seminarausarbeitungen

Hier werden die Ausarbeitungen der in der Seminarphase gehaltenen Referate kurz zusammengefasst um einen Eindruck über die Grundlagen der Projektarbeit zu geben. Im wesentlichen handelt es sich hierbei um Referate aus den Bereichen Wissensrepäsentation, Wissensverarbeitung, logische Programmierung und Logik.

2.2 Arbeitsgebiete der Kleingruppen

Zu Beginn der eigentlichen Progjektgruppen Arbeit, nach Beendigung der Seminarphase, teilte sich die Gruppe zuerst in 3 Kleingruppen auf, die parallel zu der eigentlichen Projektgruppen Arbeit an einem Teilprojekt arbeiteten. Die Kleingruppen Arbeit diente dem Zweck, sich mit der theoretischen

Materie vertraut zu machen. Jede Gruppe hat sich ein konkretes praktisches Beispiel gesucht. Anhand dieses Beispiels machten sich die Gruppen, unabhängig von einander, Gedanken um eine geeignete Wissensrepräsentation, die sowohl auf Unsicherheit als auch auf Unschärfe basiert. Es haben sich drei Kleingruppen aus folgenden Bereichen herauskristalliert: Autodiagnosesysteme, Bauernregeln und Spiele.

2.3 Der Wissensrepräsentationsformalismus

Zur Repräsentation von vagem und unsicheren Wissen wird eine mehrwertige, mehrsortige Prädikatenlogik verwendet, deren Formeln mit sogenannten Labels markiert werden. Für die Wissensrepräsentation verwenden wir vier Logiken. Die Lukasiewicze, Produkt-, Gödel- und die Leesche Logik.

Grob gesehen ist ein Label eine Relation (T, D), über den Mengen T, D,wobei:

- T = Menge der Wahrheitswerte
- D = Menge der Unsicherheitwerte

Für näheres siehe Dissertation von Stephan Lehmke:

„Logic which allow Degrees of Truth and Degrees of Validity“

2.4 Wissensverarbeitung

Die Spezifikation wurde in die drei Bereiche, Pre-Processing, Processing und Post-Processing aufgeteilt.

- Pre-Processing behandelt, die möglichen Arbeitsschritte und Funktionen vor der Verarbeitung einer Wissensbasis durch das Programm, z.B.über die GUI
- Processing ist das die eigentliche Verarbeitung der Wissensbasis, z.B. durch Forward-Chaining, Backward-Chaining oder Resolution
- Post-Processing behandelt die Auswertung und Darstellung der Verarbeitung.

2.5 Ein Beispiel

Als Grundlage für ein kleines Beispiel, für die vage und unsichere Wissensrepräsentation und -verarbeitung dienen die Ergebnisse der „Bauernregeln“-Gruppe, die sich mit dem gleichnamigen Buch („Bauernregeln“) von Horst Malberg beschäftigt hat. Dieses Buch enthält Bauernregeln, die der Autor

anhand von meteorologischen Daten der letzten 50 bis 100 Jahre auf ihre Gültigkeit hin untersucht und für die er einen tatsächlichen Zusammenhang mit dem Wettergeschehen festgestellt hat. Dabei liefern die zum Teil angegebenen Eintrittswahrscheinlichkeiten der Bauernregeln die Unsicherheit, die natürlichsprachlichen Ausdrücke wie „kalt“, „nass“, „freundlich“ oder „gut“ die Vagheit. Da die Bauernregeln nur recht grobe Angaben machen, wurden sie in dem Buch als Abweichungen vom Mittelwert interpretiert, d.h. die Worte „kühl“, „kalt“ oder „frostig“ bezeichnen alle eine Temperatur, die unter dem langjährigen Durchschnitt des jeweiligen Zeitraumes liegt, was sich in einer Vereinfachung der Wissensrepräsentation niederschlägt.

2.6 Das weitere Vorgehen

Geplant wird die Erweiterung/Verbesserung der Implementierung. Weiterhin soll die Wissensrepräsentation und Wissensverarbeitung von der Aussagenlogik hin zur Prädikatenlogik erweitetert werden, um damit die Ausdrucksstärke zu steigern. Zudem sollen unterschiedlicher Logiken verwendet werden.

Kapitel 3

Funktionsumfang

Das Programm Fuzzy^2 kann unscharfes und unsichers Wissen verarbeiten und auswerten. Hierzu kommt eine Kombination aus Fuzzy- und Possibilistischer Logik zur Anwendung, die auf der Bewertung von Formeln durch Label, die Vagheit und Unsicherheit repräsentieren, basiert. Mit Hilfe des Programmes kann man eine Wissensbasis erstellen, bearbeiten und Anfragen an diese stellen.
Eine Wissensbasis wird vom Programm mit Beschreibung als Textdatei gespeichert. Diese Textdatei wird beim Laden vom Programm geparsed und in eine Baumdarstellung zur internen weiterverabeitung umgewandelt. Eine Wissensbasis kann natürlich auch wieder als Textdatei gespeichert werden.
Eine Formel der Wissensbasis besteht aus der eigentlichen logischen Formel und einer Bewertungsfunktion, dem Label. Diese Label kann mit Hilfe eines Editorfensters komfortabel eingegeben und bei Bedarf verändert werden. Dies geschieht, indem man einzelne Stützpunkte per Maus oder Tastatur eingibt, bzw. verändert.
Dieses Bewertungslabel wird dann einer logischen Formel zugewiesen. Dem Programm liegt eine mehrsortige Prädikatenlogik zu Grunde. Die Formel darf daher aus Konstanten, Variablen, Formelvariablen, logischen Konnektoren und logischen Quantoren bestehen, für die gegebenenfalls noch eine Sorte spezifiziert werden kann.
Zur Auswertung kann eine Anfrage an das Programm gestellt werden. Das Programm versucht dann diese Formel aus der Wissensbasis per Forwardchaining und Modus Ponens abzuleiten. Bisher unterstützt das Programm nur die Lukasiewiczsche Logik und auch nur dementsprechende Axiome. Dank Modularität lässt sich sie Axiomedatei austauschen und so sind theoretisch auch andere Logiken verwendbar.
Zur Verarbeitung der Anfrage stehen verschiedede Heuristiken zur Verfügung. Man kann die Länge von Formeln begrenzen, die zur Verarbeitung herangezogen werden und so die Verarbeitungszeit einschränken. Allerdings ist es dadurch auch möglich, dass eine vorhandene Lösung nicht gefunden wird.

Ausserdem kann man Formelvariablen bewerten und so forcieren, dass erstmal Formeln mit weniger Formelvariablen bearbeitet werden. Auch hierdurch kann eine eventuell Vorhandene Lösung nicht gefunden werden.
Während der Verarbeitung wendet das Programm abwechselne Axiome und Modus Ponens an. Die einzelnen Verarbeitungsschritte werden mittels eines Fortschrittbalkens dargestellt. Ausserdem wird dem Benutzer die Möglichkeit gegeben, die Verarbeitung durch einen Button vorzeitig anzubrechen.

Kapitel 4

Theorie und Algorithmus

4.1 Einleitung

In unserem Algorithmus haben wir die Łukasiewiszsche Erweiterung der Prädikatenlogik realisiert. Deren Syntax und Semantik wird im folgenden nochmals erläutert (4.2 und 4.3). In 4.4 erläutern wir kurz die Funktionsweise des Forward Chaining und in 4.5 dessen Implementierung. 4.6 beschreibt Idee und Implementierung des Stützpunktalgorithmus, den wir zur Kombination der Label verwenden.
Die Abschnitte 4.2, 4.3 und 4.6 sind größtenteils dem Zwischenbericht entnommen.

4.2 Mehrwertige Prädikatenlogik

4.2.1 Syntax der mehrwertigen, mehrsortigen Prädikatenlogik:

Zum Alphabet gehören:

- die technischen Hilfszeichen) und, und (
- eine Menge von Konstanten zur Bezeichnung der Quasiwahrheitswerte aus dem Intervall [0,1] der reellen Zahlen
- eine Menge von Konstanten zur Bezeichnung bestimmter Objekte des Individuenbereichs
- eine abzählbare Menge von Prädikatensymbolen
- eine abzählbaren Menge von Funktionssymbolen
- die Menge der Quantoren $\{\exists, \forall\}$, sowie bei Bedarf noch weitere Quantoren

- eine nichtleere (oft endliche) Menge S von Sorten
- für jede Sorte $s \in S$ eine abzählbar unendliche Menge von Variablen, notiert u^S, v^S, w^S,... mit oder ohne Indizes
- die Menge der einstelligen Junktoren $\{\neg\}$
- die Menge der zweistelligen Junktoren $\{\wedge, \vee, \Rightarrow\}$

4.2.2 Definition von Sorten

Sei S eine Menge von Sorten. Eine Mengenfamilie $A = A_{ss\in S}$ heißt S-sortierte Menge. (Eine Mengenfamilie ist eine Abbildung, die jedem Element einer Indexmenge (hier S) eine Menge zuordnet.) Ist $w = s_1...s_n \in S^+$, dann schreiben wir A_w anstelle von $A_{s_1} \times ... \times A_{s_n}$. Sind A und B S-sortige Mengen, dann heißt eine Familie $R = \{R_s \subseteq A_s \times B_s\}_{s\in S}$ von Relationen S-sortierte (binäre) Relation. Sind A und B S-sortige Mengen, dann heißt eine Funktion f: A $\rightarrow$ B S-sortiert, wenn es für alle $s \in S$ eine Funktion $f_s : A_s \rightarrow B_s$ existiert mit $f_s(a) = f(a)$ für alle $a \in A_s$. Ist $s_1, ..., s_n \in S$ und $w = s_1, ..., s_n$, dann wird die Funktion $f_w : A_w \rightarrow B_w$ definiert durch $f_w(a) = (f_{s_1}(a_1), ..., f_{s_n}(a_n))$ für alle a = $(a_1, ..., a_n) \in A_w$.

4.2.3 Definition einer mehrsortigen Signatur

Eine (mehrsortige) Signatur $\Sigma = (S, F, R)$ besteht aus einer Menge S von Sorten und zwei S^+-sortierten Mengen F von Funktionssymbolen und R von Prädikaten oder Relationssymbolen. Die Elemente von $F_s, s \in S$, heißen Konstanten. Anstelle von

$f \in F_s$ bzw. $g \in F_{s_1...s_ns}$ bzw. $r \in R_{s_1...s_n}$

schreiben wir

$f :\rightarrow s \in \Sigma$ bzw. $g : s_1...s_n \rightarrow s \in \Sigma$ bzw. $r : s_1...s_n \in \Sigma$ oder
$g : s_1 \times ... \times s_n \rightarrow s \in \Sigma$ bzw. $r : s_1 \times ... \times s_n \in \Sigma$.

4.2.4 Definition von Term und Atom

Sei $\Sigma = (S, F, R)$ eine Signatur und X eine S-sortige Menge von Individuen-Variablen. Die S-sortige Menge $T_\Sigma(X)$ der Σ-Teme und die Menge $At_\Sigma(X)$ der $\Sigma - Atome$ sind induktiv definiert:

- Für alle $s \in S$ ist $X_s \subseteq T_\Sigma(X)_s$.
- Für alle $w \in S^*$, $s \in S, f : w \rightarrow s \in \Sigma$ und $t \in T_\Sigma(X)_w$ ist $f(t) \in T_\Sigma(X)_s$.

- Für alle $w \in S^+$, $r : w \in \Sigma$ und $t \in T_\Sigma(X)_w$ ist $r(t) \in At_\Sigma(X)$.

Ein variablenfreier Term heißt Grundterm. Die Menge der Σ-Grundterme wird mit T_Σ bezeichnet. Ist Σ die abstrakte Syntax einer CF-Grammatik G, dann heißen die Σ-Grundterme auch Syntaxbäume von G. Ein variablenfreies Atom heißt Grundatom oder Fakt.

4.2.5 Definition einer Formel

Sei Σ = (S, F, R) eine Signatur und X einer S-sortierte Variablenmenge. Die Menge der (prädikatenlogischen) Σ-Formeln (über X) ist induktiv definiert:

- Jedes Σ-Atom ist eine Σ-Formel.
- Für alle Σ-Formeln φ, ψ und $x \in X$ sind auch $\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \Rightarrow \psi, \varphi \Leftrightarrow \psi, \forall x\varphi$ und $\exists x\varphi$

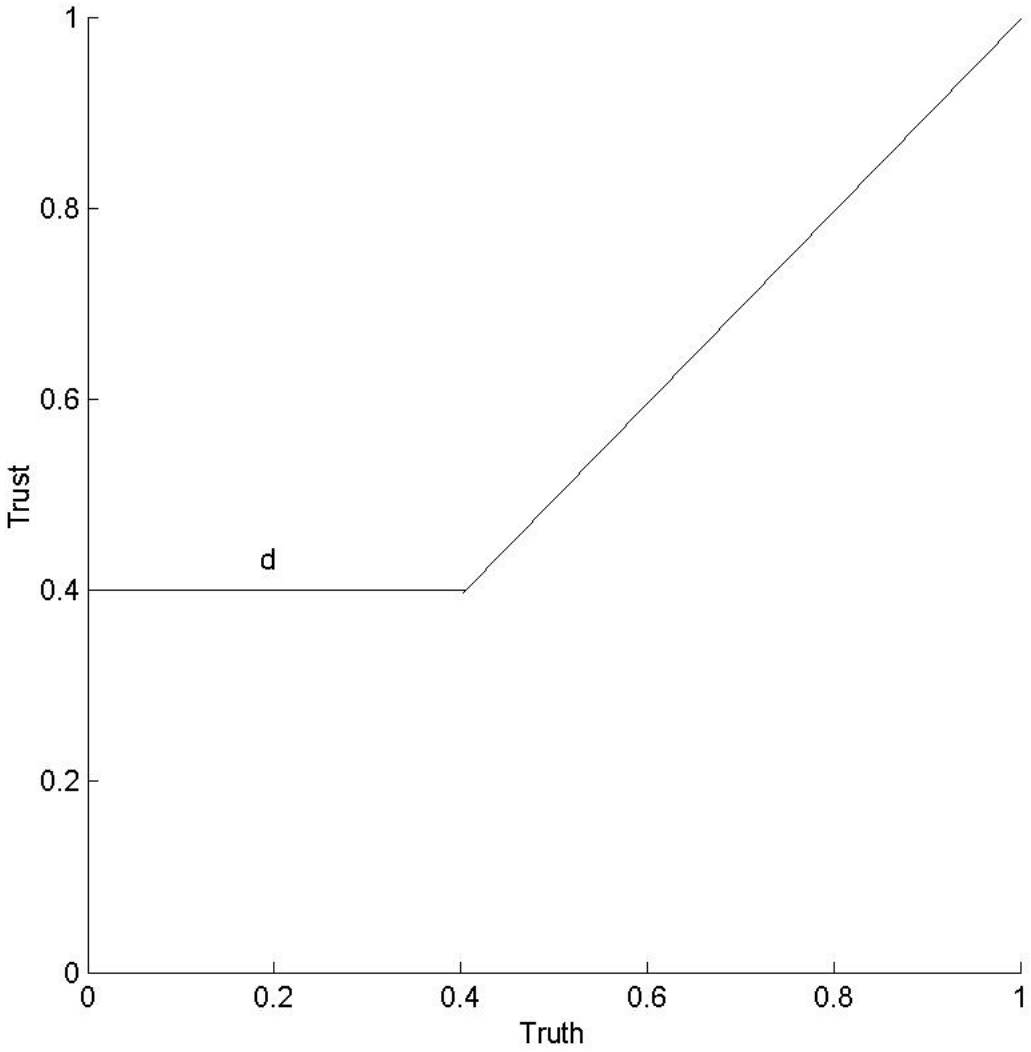
Jede Formel kann ein Label besitzen, wenn dem so ist, dann heißt sie markierte Formel.

4.2.6 Was ist ein Label

Grob gesehen ist ein Label eine Relation (T, D), über den Mengen T, D, wobei:

- T = Menge der Wahrheitswerte
- D = Menge der Unsicherheitwerte

Ein Label kann als Markierung einer Formel bzw. Aussage gesehen werden, die eine Bewertung bezüglich Vagheit und Unsicherheit liefert. Das Label kann als zweidimensionaler Graph dargestellt werden, siehe zum Beispiel Abbildung 4.2.6. Für näheres siehe Dissertation von Stephan Lehmke „Logic which allow Degrees of Truth and Degrees of Validity“



Ein mögliches Label

4.3 Axiome

Für die Wissensverarbeitung verwenden wir die Łukasiewicze Logik.

4.3.1 Prädikatenlogische Basislogik

Axiomensystem:

- (A1) $(\varphi \to \psi) \to ((\psi \to \chi) \to (\varphi \to \chi))$
- (A2) $(\varphi \& \psi) \to \varphi$
- (A3) $(\varphi \& \psi) \to (\psi \& \varphi)$
- (A4) $(\varphi \& (\varphi \to \psi) \to (\psi \& (\psi \to \varphi))$
- (A5a) $(\varphi \to (\psi \to \chi)) \to ((\varphi \& \psi) \to \chi)$
- (A5b) $((\varphi \& \psi) \to \chi) \to (\varphi \to (\psi \to \chi))$
- (A6) $((\varphi \to \psi) \to \chi) \to (((\psi \to \varphi) \to \chi) \to \chi)$
- (A7) $\overline{0} \to \varphi$

Die Axiome A1 bis A7 kann man auch als ein Axiomensystem der Aussagenlogik betrachten. Die nachfolgenden Axiome beziehen sich auf die Quantoren der Prädikatenlogik.

- ($\forall$1) $(\forall x)\varphi(x) \to \varphi(t)$ (t ist einsetzbar für x in $\varphi(x)$)
- ($\exists$1) $\varphi(t) \to (\exists x)\varphi(x)$ (t ist einsetzbar für x in $\varphi(x)$)
- ($\forall$2) $(\forall x)(\nu \to \varphi) \to (\nu \to (\forall x)\varphi)$ (x nicht frei in ν)
- ($\exists$2) $(\forall x)(\varphi \to \nu) \to ((\exists x)\varphi \to \nu)$ (x nicht frei in ν)
- ($\forall$3) $(\forall x)(\varphi \vee \nu) \to ((\forall x)\varphi \vee \nu)$ (x nicht frei in ν)

4.3.2 Łukasiewicze Prädikatenlogik

Als Verknüpfungen haben wir die Łukasiewiecze t-Norm und die daraus enstandene Implikation:

- Łukasiewiecze t-Norm: $x * y = max(0, x + y - 1)$
- Łukasiewiecze Implikation: $x \to y = 1 - x + y$

Die Łukasiewiczsche Prädikatenlogik läßt sich bewiesenermaßen nicht vollständig axiomatsieren, jedoch gibt es Ansätze die einer Axiomatisierung schon sehr nahe kommen. Man nehme vorherige Axiome der Basislogik sowie die folgenden Łukasiewiczen Axiome.

- (Ł1) $\varphi \to (\psi \to \varphi)$
- (Ł2) $(\varphi \to \psi) \to ((\psi \to \chi) \to (\varphi \to \chi))$
- (Ł3) $(\neg\varphi \to \neg\psi) \to (\psi \to \varphi)$
- (Ł4) $((\varphi \to \psi) \to \psi) \to ((\psi \to \varphi) \to \varphi)$

Näheres kann in der Seminarausarbeitung 'Mehrwertige Logik' nachgelesen werden.

4.4 Wissensverarbeitung mit Forward-Chaining

Ohne spezielle Anfrage werden beim Forward Chaining, aus den Fakten der Wissensbasis alle weiteren möglichen Fakten und Regeln abgeleitet. Liegt eine Anfrage vor, werden solange Formeln abgeleitet, bis die gewünschte Formel gefunden ist.

Die Ableitung neuer Formeln geschieht durch Anwendung des Modus-Ponens oder der Axiome.

Forward-Chaining, als Breitensuche gesehen, läßt die Wissensbasis sehr schnell anwachsen. Für eine effiziente Suche nach speziellen Anfragen ist es sinnvoll Heuristiken anzuwenden um den Suchraum einzuschränken. Eine uns sinnvoll erscheinende Heuristik, ist die Einschränkung auf kürzere Formeln, d.h. Formeln werden nur zur Weiterverarbeitung herangezogen, wenn sie eine gewisse Länge nicht überschreiten. Zusätzlich können bestimmte Terme, von denen man annimmt, daß sie nicht zur Ergebnisfindung beitragen, mit Strafkosten durch höhere Länge bewertet werden.

4.5 Implementierung

4.5.1 Forward-Chaining

In der Kasse FCPL befinden sich mehrere process-Methoden mit unterschiedlichen Aufrufparametern. Diese Methoden steuern die abwechselnde Anwendung des Modus-Ponens und der Axiome. Prinzipiell werden in einer Endlos-Schleife abwechselnd die Methoden InfTool.axiomeFuerAlleAnwenden() und InfTool.modusPonensFuer AlleAnwenden() aufgerufen. Falls die bei der Anfrage gestellte Formel erzeugt wird, sorgt eine Ausnahmenbehandlung für das Verlassen der Schleife. Die Klasse FCPLThread versieht die process-Methode mit einem eigenständigen Thread, so daß sie nebenläufig gestartet werden kann.

4.5.2 Modus-Ponens

Die Anwendung des Modus-Ponens wird in der statischen Methode InfTool.modusPonensFuerAlleAnwenden() realisiert. Diese versucht durch Einzelaufrufe der Methode modusPonens(MarkFormel) für jede einzelne Formel der Wissensbasis den Modus-Ponens anzuwenden.

```
modusPonens(MarkFormel)
if Formel keine Implikation
     Abbruch
else
     Suche Prämisse in WB
     if nicht gefunden
          if konjugierte Formel
```

```
            suche einzelne Komponenten in WB
            if Komponenten vorhanden
                erzeugePrämisse
                addToKB
        if quantorisierte Formel
            teste ob Quantor erfüllt
            if erfüllt
                erzeugePrämisse
                addToKB
    if keine Prämisse
          Abbruch
    else
          Verknüpfung des Prämissenlabels mit dem Formellabel
          addToKB Konklusion mit der Verknüpfung
```

4.5.3 Axiome

Die Anwendung der Axiome wird in der statischen Methode InfTool.axiomeFuerAlleAnwenden(Vector ax) realisiert. Axiome liegen teilweise als Formeln im Axiomvektor ax. Durch Einzelaufrufe der Methode axiomAnwenden (KonjFormel ax1, MarkFormel frm) wird für jede einzelne Formel der Wissensbasis versucht jedes Axiom anzuwenden.

```
axiomAnwenden(KonjFormel ax1, MarkFormel frm)
if gleiche Struktur
     unifiziere Prämisse des Axiom mit Formel
     und ersetze Formelvariablen des Axioms entsprechend
     addToKB Konklusion des Axioms mit Label der Formel
```

Zusätzlich gibt es hartcodierte Axiome, welche in den Methoden axiomGeneralisierung (MarkFormel mf, Konstante k) und axiomSubstitution (MarkFormel mf, Konstante k) realisiert sind. Generalisierung implementiert Axiom ($\exists 1$) und Substituion implementiert ($\forall 1$) (siehe 4.3.1).

```
axiomSubstitution(MarkFormel mf, Konstante k) throws
 GefundenException{
      if Formel allquantorisiert
            bestimme an Quantor gebundene Variable
            varDurchKonstErsetzen(Variable, Konstante)
            addToKB Ersetzung mit Label der Formel
```

```
axiomGeneralisierung (MarkFormel mf, Konstante k)
throws GefundenException{
     erzeuge neue, eindeutige Variable
     if Konstante kommt in Formel vor
          konstDurchVarErsetzen(a,v,k)
          binde Variable an ExistenzQuantor
          addToKB Existenzquantorformel mit Label
          der Formel
```

Weitere Details sind dem Programmcode bzw. dessen Kommentaren zu entnehmen.

4.5.4 Vergleichen und Unifizieren von Formeln

Der entscheidene Punkt des Alghorithmus ist das Vergleichen und „Unifizieren“ der Formeln. „Unifizieren“ bedeutet hier, dass durch erlaubte Ersetzungen zwei Formeln auf die gleiche Form gebracht werden. Sind A und B Formelteile und β eine Formelvariable, so können die Formeln $A \rightarrow B$ und $A \rightarrow \beta$ unifiziert werden, indem β durch B ersetzt wird.
Dieses Unifizieren ist beim Vergleichen der Formel dann erwünscht, wenn bei der Anwendung des Modus Ponens die Prämisse in der Wissensbasis gesucht wird. Wird beim Hinzufügen einer Formel in der Wissensbasis gesucht, ob eine identische Formel bereits existiert, ist es natürlich nicht erwünscht. Daher werden verschiedene Methoden zum Formelvergleich bereitgestellt.

Zunächst betrachten wir den einfachen Fall (ohne Ersetzen von Formelvariablen):
Der Vergleich zweier Formeln erfolgt über einen rekursiven Vergleich der Formelteile. Zwei Formeln sind gleich, wenn sie vom gleichen Typ sind (also z.B. konjugierte Formeln mit gleichem Operator) und die Teilformeln gleich sind. Wir steigen also rekursiv parallel in beiden Formelbäumen ab, vergleichen an jeder Stelle ob der Formeltyp identisch ist; abhängig vom Typ müssen weitere Bedingungen erfüllt sein, so müssen z.B. Operatoren, Quantoren, Prädikate, Konstanten, Stelligkeit von Funktionen und die Sorten von Variablen übereinstimmen. Der Vergleich von Variablen ist aber mit dem Vergleich der Sorte nicht getan; zusätzlich muss deren Stellung innerhalb der Formel identisch sein (Betrachtung eines einfachen Falles: sind P und Q einstellige Prädikate und a, b und c Variablen der gleichen Sorte, sind die Formeln $P(a) \rightarrow Q(b)$ und $P(c) \rightarrow Q(c)$ nicht identisch). Dies wird durch zwei Vektoren gewährleistet, in die alle Variablen, die während des Vergleichs gefunden werden, gespeichert werden: Beide Formeln erhalten einen zunächst leeren Vektor, in denen nun jeweils die gefundenen Variablen vermerkt werden. Gefundene Variablen müssen in beiden Vektoren die glei-

che Position haben (also zunächst, da beide Vektoren leer sind, „-1“ = „nicht vorhanden“). Dann sind sie unifizierbar.
Dieses Verfahren wird für Formelvariablen in der gleichen Weise benutzt.

Sollen nun beim Vergleich ggf. Formelvariablen ersetzt werden, geschieht dieses durch zwei weitere Vektoren. Wird in einer Formel eine Formelvariable (im Folgenden α) gefunden und in der anderen Formel steht an dieser Stelle ein anderer Formelteil (im Folgenden A), kann die Formelvariable durch diesen Formelteil ersetzt werden. Zunächst werden α und A in den jeweiligen Vektor an die jeweils gleiche Stelle geschrieben. Erst nach dem kompletten rekursiven Durchlauf und sonstiger Gleichheit werden anschließend die Ersetzungen gemäß der Vektoren durchgeführt. Dabei wird darauf geachtet, dass, falls eine Formelvariable mehrfach vorkommt, die jeweilige Ersetzung übereinstimmt.
Hier soll noch angemerkt werden, dass bei der Anwendung des Modus Ponens nur die Prämisse in der Wissensbasis gesucht wird, die Ersetzung aber ggf. auch in der Konklusion durchgeführt werden muss (und durchgeführt wird) - falls sich in Prämisse und Konklusion identische Formelvariablen wiederfinden.

4.5.5 Behandlung von Quantoren

Die Auflösung von Quantoren geschieht in der erfuellt-Methode der Quantoren-Objekte. Übergeben wird eine konjugierte Formel (Implikation), welche auf der linken Seite als Prämisse eine quantorisierte Formel enthält. In dieser quantorisierten Formel werden für die vom Quantor abhängige Variable sukzessive alle Individuen der zugehörigen Sorte eingesetzt und überprüft, ob diese Formel in der Wissensbasis vorhanden ist. Abhänging davon ob es sich um einen Existenz- oder Allquantor handelt, wird folgendermaßen vorgegangen:

4.5.5.0.1 Allquantor Um den Allquantor zu erfüllen müssen alle Individuen ein positives Resultat liefern. Dies enspricht einer Konjunktion über alle Individuen eingesetzt in die quantorisierte Formel. Die Bewertung der quantorisierten Formel ergibt sich aus dem FLabel des schlechtesten Faktors bei der Einsetzung. Diese Bewertung wird zurückgegeben und dient als Bewertung für die Konklusion der übergebenen Implikation.

4.5.5.0.2 Existenzquantor Um den Existenzquantor zu erfüllen muss lediglich ein Individuum ein positives Resultat liefern. Dies enspricht einer Disjunktion über alle Individuen eingesetzt in die quantorisierte Formel. Die Bewertung der quantorisierten Formel ergibt sich aus dem FLabel des besten Summanden bei der Einsetzung. Diese Bewertung wird zurückgegeben.

4.6 Labelkombination mit dem Stützpunkt Algorithmus

Die Unschärfe und Unsicherheit stellen wir durch Label dar (siehe 4.2.6). Bei der Axiomanwendung erhält die neue Formel das Label der Formel, auf die das Axiom angewendet wurde. Bei der Anwendung des Modus Ponens werden die Label der beteiligten Funktionen undverknüpft. Diese Kombination der Label ist nicht trivial, der Stützpunktalgorithmus der dieses realisiert wird im Folgenden beschrieben.

4.6.0.1 Theorie

Um mit Hilfe der in diesem Kapitel spezifizierten Logik Folgerungen machen und Formeln verknüpfen zu können, muss man zwei Bewertungslabel zu einem semantisch korrekten Folgelabel verknüpfen können. Wie Stephan Lehmke in seiner Dissertation gezeigt hat, lässt sich das Ergebnislabel aus den zwei Ursprungslabeln durch folgende Formel berechnen:

$$F\tau G(x) = Sup\{min(F(y), F(z))|y, z \in [0,1] \wedge \tau(y,z) = x\}$$

Das τ ist eine Funktion, die von der verwendeten Logik abhängt. In der Łukasiewiczschen Logik entspricht das τ der Łukasiewiczschen Konjunktion ($\wedge_{bold}$): $\tau(a,b) = max(0, a+b-1)$.
Mit Hilfe dieser Formeln kann man aus zwei Punkten der jeweiligen Ursprungslabel den resultierenden Punkt im Ergenislabel berechnen. Allerdings reicht es ja nicht aus, zwei Punkte zu kombinieren, sondern man muss ja, um zu einem vernünftigen Label zu gelangen, alle Punkte miteinander verknüpfen.

Wir betrachten im Folgenden nur Label, die stückweise linear sind, d.h. sie bestehen aus einer Anzahl von Geraden, die an Stützpunkten zusammengesetzt sind. Ein Label kann nun als Folge von Stützpunkten repräsentiert werden. Da auch das Ziellabel stückweise linear sein soll, werden nur T-Normen betrachtet, die diese Vorgabe erfüllen; hier also die (Łukasiewiczche) Bold- und ausserdem die Min-Konjunktion.

4.6.1 Die Idee des Algorithmus

Ziel des Algorithmus ist es, aus den Stützpunkten zweier Label die Stützpunkte des Ziellabels zu gewinnen. Zunächst wurde angenommen, dass es ausreichen würde, paarweise alle Stützpunkte anhand der Formel (s. o.) zu kombinieren um so alle Stützpunkte des Ziellabels zu erhalten. Leider erwies sich das Problem als etwas komplizierter.

1. Es müssen Geraden betrachtet werden
Wenn wir als Ergebnis nur eine Folge von Punkten produzieren, die in keinem Zusammenhang zueinander stehen, können wir leider nicht eindeutig bestimmen wie das Label aussieht. Es ist nämlich möglich, dass einer dieser Punkte unterhalb einer Gerade liegt, die zwei andere Stützpunkte verbindet; wir haben keine Information, ob diese Gerade existiert.
Daher werden nicht nicht Punkte sondern sämtliche Geraden paarweise kombiniert. Die Kombination zweier Geraden erfolgt folgendermaßen:
$a1$ und $a2$ seien Start- und Endpunkt der ersten Geraden.
$b1$ und $b2$ seien Start- und Endpunkt der zweiten Geraden.
Daraus können wir 5 Geraden erzeugen:

- $a1 \circ b1 \longrightarrow a1 \circ b2$
- $a1 \circ b1 \longrightarrow a2 \circ b1$
- $a1 \circ b1 \longrightarrow a2 \circ b2$
- $a1 \circ b2 \longrightarrow a2 \circ b2$
- $a2 \circ b1 \longrightarrow a2 \circ b2$

Hierbei entspricht „$\circ$“ der Verknüpfung zweier Punkte mittels der Formel.
Diese Geraden sind korrekt, das heisst neben den Endpunkten können auch alle anderen Geradenpunkte aus den Ursprungslabeln erzeugt werden (Ausnahme: Siehe 2).
Wir bilden nun auf diese Weise alle möglichen Geraden und erhalten einen Graphen. Das Ziellabel ist die „obere Hülle“ dieses Graphen, da ja laut der Formel für jeden Punkt der maximale erzeugbare Wert gewählt werden muss.

2. Das betrachtete Intervall muss in den negativen Bereich ausgedehnt werden
Leider führt der Ansatz bei der Bold-Konjunktion noch nicht zum Ziel. Die Bold-Konjunktion lautet $f(a, b) = max(0, a + b - 1)$. Unsere vorige Idee ist jedoch nur korrekt für Geradengleichungen. Wir können Abhilfe schaffen, indem wir die Bold-Konjuktion „zunächst“ als Geradengleichung $f(a, b) = a + b - 1$ auffassen (wir können entweder a oder b als fest ansehen). Dadurch erhalten wir Punkte mit negativer x-Koordinate und somit auch Geraden bis in den negativen Bereich, also bis hin zu -1. Der wichtige Effekt ist, dass wir Geraden mit Startpunkt im negativen und Endpunkt im positiven Bereich erhalten; diese Geraden sind wichtig, denn der Geradenteil im positiven Bereich kann Teil der oberen Hülle sein. Anschliessend können wir dann an der y-Achse einen Schnitt machen. Für alle Punkte rechts von der y-Achse haben wir korrekte Werte erhalten; die Werte links von der y-Achse interessieren uns nicht mehr.

3. Es müssen zusätzliche Stützpunkte betrachtet werden
Der Algoritmus ist korrekt, wenn wir alle notwendigen Stützpunkte des Ziellabels erhalten und sämtliche nötigen korrekten Geraden zwischen diesen Stützpunkten erzeugt haben. Unsere Hoffnung war zunächst, dass alle notwendigen Stützpunkte aus den Stützpunkten der Ursprungslabel entstehen. Leider ist das nicht immer der Fall. Es ist uns aber gelungen, die Punkte der Ursprungslabel zu identifizieren, aus denen die notwendigen Punkte erzeugt werden können. Dies sind:

- die Stützpunkte der Ursprungslabel
- die Punkte an den x-Stellen, an denen das jeweils andere Ursprungslabel Stützpunkte hat
- die Punkte, deren x-Stellen den y-Werten von Stützpunkten eines der Label entsprechen
- die Punkte mit x-Stellen $x1$, die bzgl. des Intervalls [0,1] „invers“ zu den x-Stellen $x2$ anderer Stützpunkte eines der Label sind, d.h. Stellen mit $x1 = 1 - x2$.

Werden in den Ursprungslabeln an diesen Stellen zusätzliche Punkte eingefügt, lassen sich daraus fast alle notwendigen Stützpunkte des Ziellabels gewinnen - nämlich alle, die zur Erzeugung der notwendigen Geraden notwendig sind. Die Schnittpunkte dieser Geraden können dann ebenfalls Stützpunkte des Ziellabels sein. Schnittpunkte von Geraden müssen also ebenfalls in den Graphen eingefügt werden, die entsprechenden Geraden werden unterteilt.
Dieses Vorgehen sollte korrekt sein für die Bold- und die Min-Konjunktion. Ein Beweis konnte bisher nicht erbracht werden, die empirischen Testergebnisse sind aber vielversprechend.

4. Die Erzeugung aller notwendigen Geraden
Die Erzeugung aller korrekten Geraden ist durch das Einfügen zusätzlicher Stützpunkte etwas komplizierter geworden.
Prinzipiell bleibt das Vorgehen gleich: alle Geraden eines Labels werden mit allen Geraden des anderen Labels wie oben beschrieben kombinert. Es müssen allerdings nun weiterhin auch die Geraden betrachtet werden, die durch eingefügte Stützpunkte 'unterbrochen' worden sind.
Auf diese Weise werden alle korrekten Geraden erzeugt, und die obere Hülle des entstehenden Graphen entspricht dem gesuchten Ziellabel. Diese obere Hülle lässt sich durch einen Suchlauf im Graphen ermitteln. Die Stützpunkte der Geraden, die diese obere Hülle bilden, sind die Stützpunkte des Ziellabels. Stützpunkte, die Geraden gleicher Steigung verbinden, können an dieser Stelle entfernt werden.

4.6.2 Implementierung

Der Algorithmus wurde in Form des „Labelkombinators“ umgesetzt. Diese Umsetzung soll hier kurz beschrieben werden.

Die Repräsentation des Labels

Ein Label wird durch ein *FLabel*-Objekt repräsentiert, das im Wesentlichen aus einer Stützpunktliste (*StPktList*: ein erweiterter Vector) besteht. Durch *add*-Methoden können neue Stützpunkte hinzugefügt werden. Die Stützpunkte werden durch *StPkt*-Objekte repräsentiert.
Zum Kombinieren von Labeln mittels Bold- oder Min-Konjunktion werden die Methoden *mergeBold* und *mergeMin* bereitgestellt. Als Argument erhalten sie das Label, mit dem das Label selbst kombiniert werden soll. Das Ziellabel wird zurückgegeben.
Das Minimum zweier Label liefert analog die Methode *min*.
Diese genannten Methoden erzeugen ein Objekt einer Control-Klasse (*Label Kombinator*), welches die Operationen durchführt. Sie sind in *FLabel* integriert, um eine möglichste kleine und einfache Schnittstelle zu erhalten.

Label - Kombination

Der *LabelKombinator* ist die Control-Klasse aller Label-kombinationen. Bei der Instantiierung muss die Konjunktion übergeben werden (also Und_{bold} oder Und_{min}), anhand der die Kombination durchgeführt werden soll. Die Konjunktionen verfügen über zwei Methoden zur Berechnung der Konjunktion: *calc* liefert das korrekte definitionsgemäße Ergebnis, *calc_neg* kann (bei der Bold-Konjunktion) auch negative Werte zurückgeben (s.o.).
Die *getIt*-Methode erhält zwei Label als Argumente, kombiniert sie und gibt das Ziellabel zurück. Zunächst werden Kopien der übergebenen Label erstellt (dazu stellt *FLabel* die Methode *copy* bereit). In diese Kopien werden dann weitere Stützpunkte an den oben beschriebenen Stellen eingefügt. *FLabel* verfügt über eine Methode *getTwerte*, die diese Stellen angibt. Dann werden sehr viele Punkte und Geraden erzeugt (das Programm sollte korrekt sein, wenn tatsächlich alle korrekten Geraden, die Teil einer oberen Hülle sein könnten, erzeugt werden). Diese Punkte und Geraden werden in einen Graphen (*LGraph*) eingefügt. Anschließend wird an der Stelle x=0 temporär eine senkrechte Gerade eingefügt, das entspricht dem oben beschriebenen Abschneiden. Im Graphen wird schließlich ein Suchlauf durchgeführt, der die Stützpunkte der oberen Hülle des Graphen liefert.

Erzeugung des Graphen

Die Klasse LGraph stellt den Graphen dar, der bei der Kombination der Label entsteht. Er erhält Punkte und Geraden, die durch die *insert*-Methode eingefügt werden. Diese Methode erhält zwei Punkte als Argumente; diese Punkte und die verbindende Gerade werden in den Graphen eingefügt. Durch je eine globale Stützpunkt- und Geradenliste kann überprüft werden, ob Punkte und Geraden mit gleichen Koordinaten schon Teil des Graphen sind und somit nicht neu eingefügt werden müssen. Jede eingefügte Gerade wird sofort auf Schnittpunkte mit bereits existierenden Geraden überprüft. Werden Schnittpunkte gefunden, werden diese und die dabei entstehenden Geraden ebenfalls in den Graphen eingefügt.
Entscheidend für die Graph-struktur sind nicht nur die Klasse *LGraph* die den gesamten Graphen repräsentiert sondern vor allem auch die einzelnen *StPkt*-objekte, die Listen aller eingehenden und ausgehenden Geraden enthalten. Anhand dieser Listen kann dann der Suchlauf durchgeführt werden. Das geschieht durch die $getPfad$- Methoden der Punkte und Geraden.
Der Suchlauf beginnt am Punkt (0,0). Um die obere Hülle zu erhalten, muss an jedem Punkt die ausgehende Gerade mit maximaler Steigung gewählt werden. *GeradenList* stellte eine entsprechende Methode (*getBeste*) zur Verfügung. Auf diese Weise kann man oben am Graphen langlaufen bis man zum Punkt (1,1) gelangt, der keine ausgehenden Geraden mehr hat. Beim Rücklauf wird an jedem Punkt des Pfades überprüft, ob die Steigung der eingehenden Pfadgerade unterschiedlich zur Steigung der ausgehenden Pfadgerade ist. In diesem Fall ist dieser Punkt ein Stützpunkt des Ziellabels.

Label-Minimierung

Die Bestimmung eines Minimum-Label zweier Label mittels der Methode $getMin$ der Klasse $LabelKombinator$ geschieht nach einem ähnlichen Prinzip. Alle Geraden der Ursprungs-Label werden in einen LGraph eingefügt und ebenfalls sofort auf Schnittpunkte untersucht. Dann wird wie bei der Labelkombination ein Suchlauf durchgeführt, der hier die „untere Hülle“ des Graphen langläuft; dazu wird an jedem Punkt die Gerade mit minimaler Steigung gewählt. Analog zu den zuvor genannten Methoden existieren dazu $getMinPfad$-Methoden und eine $getWorst$-Methode aus $GeradenList$, die die Gerade mit minimaler Steigung zurückgibt.

4.6.3 Korrektheit und Laufzeit

- **Korrektheit des Algorithmus**
  Da bisher kein Gegenbeispiel gefunden wurde, gehen wir davon aus dass der beschriebende Algorithmus für die bold- und min-Konjuntion korrekt ist. Ein Beweis existiert leider nicht. Entscheidend ist hier, ob

tatsächlich alle Punkte, die zu Stützpunkten des Ziellabels beitragen können, identifiziert wurden.

- **Laufzeit und Korrektheit der Implementierung**
  Eine formale Laufzeitanalyse des Stützpunktalgorithmus wurde nicht durchgeführt. Es zeigt sich aber, dass sehr viele Geraden erzeugt werden und extrem viele Schnittpunkte entstehen - was zu einer unakzeptablen Laufzeit und in ungünstigen Fällen auch zu Stack Überläufen führen kann. Als Ausweg wird versucht, für das Endergebnis unnötige Geraden und Schnittpunkte zu ignorieren - jedoch ist diese Entscheidung nicht immer trivial.
  Geraden die komplett im negativen Bereich liegen sowie Schnittpunkte im negativen Bereich sind offensichtlich irrelevant und können ignoriert werden. Desweiteren werden Schnittpunkte sehr kurzer Geraden ignoriert. Dadurch konnte die Laufzeit in den Griff bekommen werden, jedoch müssen in Einzelfällen Abstriche bei der Korrektheit gemacht werden. Vermutlich können diese Probleme aber vermieden werden, wenn die Ausgangslabel „moderat“ gewählt werden (also z.B. keine krummen, zu eng beieinanderliegenden Koordinaten).

Kapitel 5

Handbuch

In diesem Kapitel soll der korrekte Gebrauch des Programms erläutert werden. Es werden dazu die einzelnen Anwendungsmöglichkeiten und Funktionen beschrieben.

5.1 Nach dem Starten

Nach dem Starten des Programms erscheint das in Abbildung 5.1 zu sehende Fenster. Dies ist das sogenannte Startfenster. Es gibt die drei Menüpunkte

Abbildung 5.1: Das Hauptfenster bzw. Startfenster

Wissensbasis, Hilfe und Beenden. Was es hiermit aufsich hat, soll im folgenden erklärt werden.

5.2 Menüpunkt Wissensbasis

Nach einem Mausklick auf den Menüpunkt Wissensbasis erscheint ein Pull-Down-Menü mit folgenden Auswahlmöglichkeiten (siehe dazu auch Abbildung 5.2)

Abbildung 5.2: Der Menuepunkt Wissensbasis im Hauptfenster

- Neu
  Eine neue Wissensbasis wird angelegt
- Laden
  Eine (zuvor gespeicherte) Wissensbasis wird geladen und geöffnet
- Speichern
  Eine Wissensbasis wird gespeichert

5.2.1 Neu

Dieser Auswahlpunkt ist z. Zt. noch ohne Funktion. Es soll aber hiermit möglich sein, eine Wissensbasis neu zu erstellen.

5.2.2 Laden

Wird im Menüpunkt Wissensbasis die Auswahl Laden gewählt, öffnet sich ein Dialogfenster, in dem die zu Ladende Wissensbasis ausgewählt werden kann (siehe Abbildung 5.3).

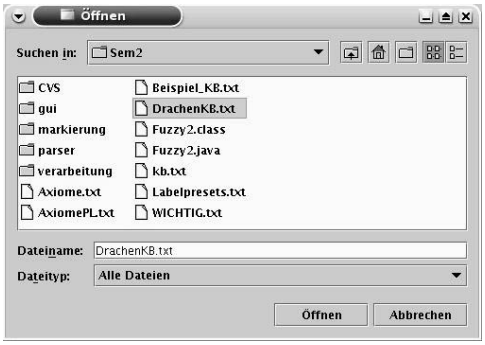


Abbildung 5.3: Laden einer vorhandenen Wissensbasis

5.2.3 Verarbeitung

Nachdem man eine Wissensbasis geladen hat, ist ein Fenster wie in Abbildung 5.4 zu sehen. Zur Veranschaulichung wurde hier bereits eine kleine Wissensbasis eingeladen. Das Fenster ist grob in drei Bereiche aufgeteil.

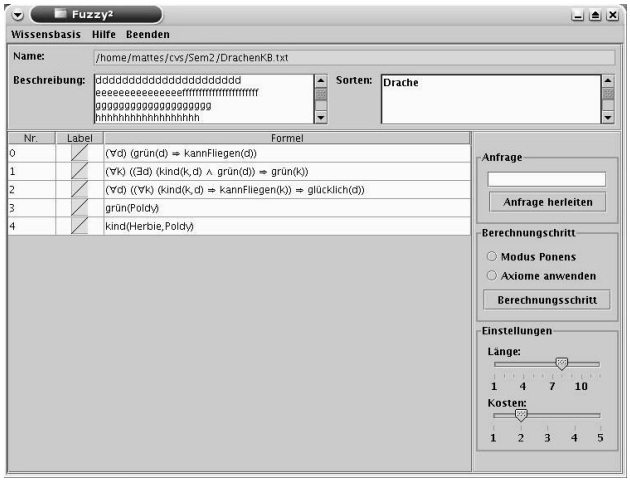


Abbildung 5.4: Ansicht nach dem Laden einer Wissensbasis

Im oberen Bereich stehen Informationen über die geladene Wissensbasis. Es ist der Pfad der geladenen Wissensbasis angegeben, eine Beschreibung der Wissensbasis und die verwendeten Sorten (im Bild nur die Sorte Drache) werden angezeigt.
Im linken (unteren) Teil sind die Formel aufgelistet. Alle Formeln sind mit 0 beginnend durchnummeriert. Neben der Nummer wird das Label der Formel in Form eines Icons angezeigt. Daneben widerrum steht die eigentliche (prädikatenlogische) Formel. Hier werden Formel auch markiert, indem man

sie einmalig mit der linken Maustaste anklickt. Um eine Formel zu löschen muß sie erst markiert werden und dann nochmals mit der rechten Maustaste angeklickt werden. Nachdem die Sicherheitsabfrage (siehe Abbildung 5.5) mit „OK“ bestätigt wurde, wird die Formel sofort entfernt.
Im rechten (unteren) Bereich des Fensters kann ausgewählt werden, was

Abbildung 5.5: Sicherheitsabfrage beim Löschen einer Formel

gemacht werden soll. Man hat die folgenden Möglichkeiten zur Auswahl:

- Man kann in dem Feld „Anfrage“ eine Formel eingeben. Nach einem Klick auf den Knopf „Anfrage herleiten“ versucht der Verarbeitungsalgorithmus die eingegebene Formel aus den vorhandenen Formeln in der Wissensbasis herzuleiten. Es erscheint daraufhin ein Fortschrittsbalken (siehe Abbildung 5.6). Kann die Anfrage aus der gegebenen Menge an

Abbildung 5.6: Fortschrittsbalken

  Formel hergeleitet werden, bricht die Verarbeitung ab, sobald die angefragte Formel gefunden wurde. Dies kann aber unter Umständen sehr lange dauern. Deshalb gibt es auch den Abbrechen-Knopf unter dem Fortschrittsbalken. Dieser bricht die Verarbeitung einfach ab, wenn man keine Lust mehr hat weiter zu warten.
  Wenn die eingegebene Formel nicht hergeleitet werden kann, läuft der Verarbeitungsalgorithmus (theoretisch) unendlich lange weiter, da es keine Möglichkeit gibt zu entscheiden, ob die angefragte Formel ableitbar ist. Man kann die Verarbeitung aber auch hier mit dem Abbrechen-Knopf beenden.

- In dem Feld Berechnungsschritt, kann eingestellt werden, was für ein Berechnungsschritt ausgeführt werden soll. Es findet aber nur immer ein Berechnungsschritt statt.
  Wird die Auswahl „Modus Ponens“ getroffen und dann der Knopf Berechnunggschritt gedrückt, wird auf die vorhandenen Formeln in der

Wissensbasis einmalig der Modus Ponens angewendet. Können neue Formeln abgeleitet werden, werden diese sofort der Wissensbasis hinzugefügt und auch im linken Teil des Fensters angezeigt.
Wird die Auswahl „Axiome anwenden“ ausgewählt und dann der Knopf Berechnungsschritt gedrückt, werden die Axiome (die in einer Extradatei gespeichert sind) auf die vorhandenen Formeln einmalig angewendet. Alle neuen Formeln, die sich mit Hilfe der Axiome herleiten ließen, werden der Wissensbasis hinzugefügt und erscheinen dann sofort im linken Teil des Fensters.

- In dem Feld „Einstellungen“ können Parameter eingetellt werden, die Einfluß darauf haben, welche Formeln abgeleitet werden. Da ein Verarbeitungsschritt bzw. eine Anfrage u. U. sehr lange dauern kann, wird hier mit Heuristiken versucht den Ablauf zu beschleunigen.
  Formeln haben (intern) eine eindeutige Länge (Länge = Anzahl der Prädikate und Formelvariablen in einer Formel). Mit den Schieberegler „Länge“ wird die maximale Länge der Formeln angegeben, bis zu der die Formel noch bei der Verarbeitung berücksichtigt wird.
  Mit dem Schieberegler Kosten kann der Wert der Formelvariablen bei der Längenberechnung der Formeln eingestellt werden.

5.2.4 Speichern

Um eventuelle Zwischenergebnisse oder Endergebnisse nicht ständig neu herleiten zu müssen, hat man die Möglichkeit die momentane Wissensbasis zu speichern. Wählt man im Menü Wissensbasis den Punkt „Speichern“ öffnet sich ein Fenster wie in Abblidung 5.7. Nachdem man einen Namen für die Wissensbasis vergeben hat, kann sie gespeichert werden.

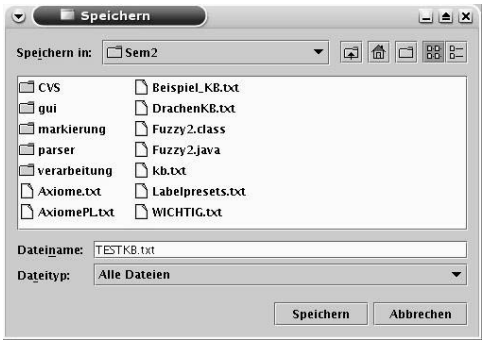


Abbildung 5.7: Speichern einer Wissensbasis

5.3 Menüpunkt Hilfe

Im Menüpunkt Hilfe können die Punkte

- Dokumentation
- Info

ausgewählt werden (siehe Abblidung 5.8).

Abbildung 5.8: Das Hilfemenue

5.3.1 Dokumentation

Wird dieser Menüpunkt angewählt, öffnet sich die Online-Dokumentation dieses Programms in einem neuen Fenster (siehe Abbildung 5.9). Diese entspricht dem (vorliegenden) Handbuch.

5.3.2 Info

Nach der Auswahl dieses Punktes, erscheint das Fenster, das auch in Abbildung 5.10 zu sehen ist. Hier wird z. Zt. nur die Programmversion angezeigt.

5.4 Formel Editor

Um eine Formel zu editieren, hinzufügen oder ihr Label zu verändern, benutzen wir den Formel Editor. Dieser erscheint automatisch, wenn man auf eine markierte Formel klickt. Dieses Fenster besteht aus den Bereichen Formel und Label, welche grün umrandet sind. Am unteren Rand befinden sich drei Buttons, mit denen Veränderungen übernommen werden können.

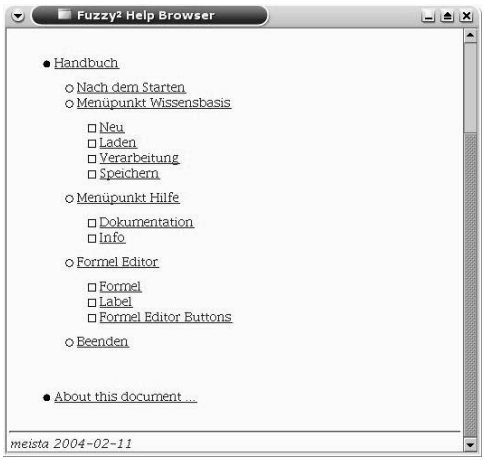


Abbildung 5.9: Die Online-Dokumentation

Abbildung 5.10: Programminformationen

5.4.1 Formel

In dem Bereich Formel (Abbildung 5.12) wird die selektierte Formel angezeigt. Wenn man die Formel verändern möchte, kann man dies hier tun. Es ist darauf zu achten, dass eine syntaktisch korrekte Formel in dem Feld steht. Ob man die Formel ändern oder ob man eine neue hinzufügen möchte, ist an dieser Stelle egal.

5.4.2 Label

Der mittlere Teil des Formel Editors ist der „Label“ Bereich. Hier werden unterschiedliche Werkzeuge zur Labelveränderung im Labelbereich (Abbildung 5.13) angeboten. Auf der linken Seite befindet sich eine grafische Eingabemöglichkeit. Auf der rechten kann man mit Hilfe von Vorlagen oder Parametereingaben das Label verändern.

- Bei der grafischen Eingabe wird die Maus als Werkzeug benutzt. Es können mit einem Linksklick neue Punkte hinzugefügt werden. Hierbei sind Punkte nur so möglich, dass das Label monoton steigt. Es ist möglich einen Punkt mit einem Linksklick zu selektieren und zu

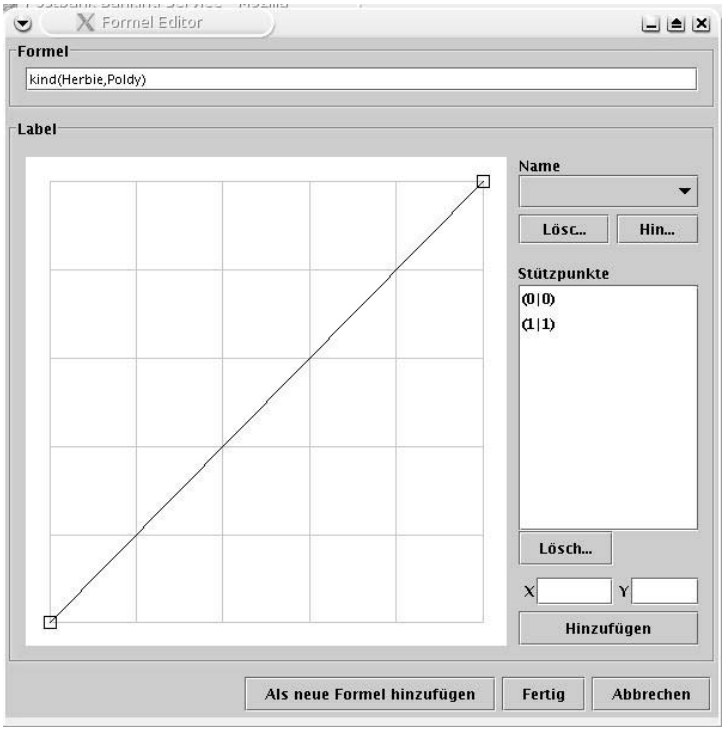


Abbildung 5.11: Der Formel Editor

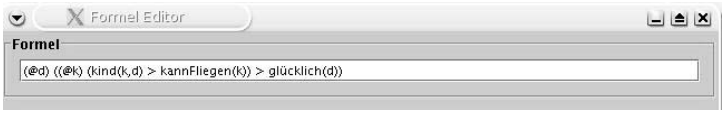


Abbildung 5.12: Bereich: Formel

verschieben. Punkte können durch einen Rechtsklick wieder gelöscht werden.
Zur besseren Orientierung wird ein Gitter mit dem Abstand 0,2 verwendet und die Position der Maus oben angezeigt.

- Rechts oben kann man mit Hilfe einer Kombobox gespeicherte Label auswählen, welche dann automatisch angezeigt werden.

- Über den Button „Löschen“ kann man eine Vorlage wieder entfernen. Zur Sicherheit erscheint ein Dialog wie in Abbildung 5.14.

- Wenn man die Vorlagen durch ein Label erweitern möchte, wählt man „Hinzufügen“ . Es öffnet sich ein Dialogfenster, in welchem man der neuen Vorlage einen Namen geben kann, siehe Abbildung 5.15

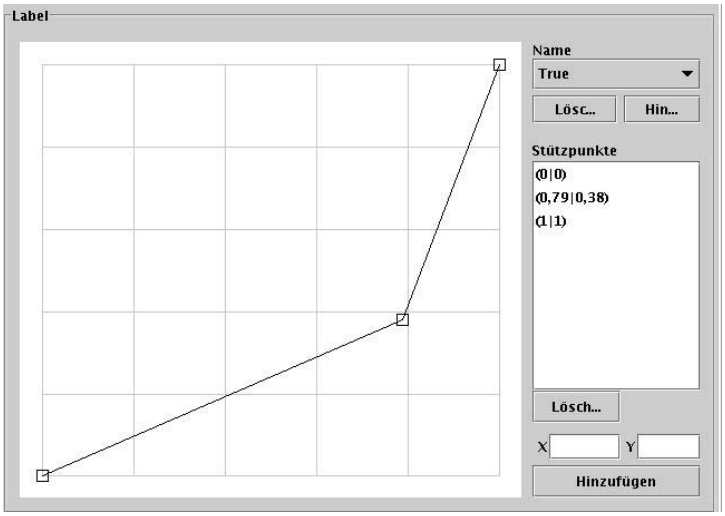


Abbildung 5.13: Bereich: Label

Abbildung 5.14: Dialog: Löschen

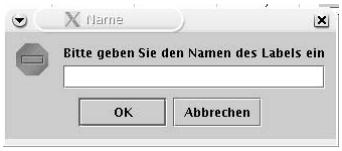


Abbildung 5.15: Dialog: Name

- Nun folgt eine Liste der Stützpunkte (Abbildung 5.16). Man kann einzelne Stützpunkte selektieren und über den nachfolgenden Button „Löschen“ entfernen.
- Die Felder „X“ und „Y“ eignen sich zur präzisen Eingabe von Stützpunkten. Über den Button „Hinzufügen“ wird das Label um den eingegebenen Stützpunkt ergänzt.

Abbildung 5.16: Liste der Stützpunkte

Abbildung 5.17: Die Buttons vom Formel Editor

5.4.3 Formel Editor Buttons

Nachdem nun Formel oder Label modifiziert worden sind, haben wir über die drei Buttons (Abbildung 5.17) folgende Möglichkeiten.

- **Als neue Formel hinzufügen:** Die Wissensbasis wird um die neue Formel bzw. das neue Label erweitert. Falls eine Fehlermeldung auftritt, ist die Formel zu überprüfen und erneut der Button zu betätigen. Nachdem eine Formel hinzugefügt worden ist, kann man im „Formel Editor“ eine weitere Formel erstellen.
- **Fertig:** Durch Betätigen von „Fertig“ wird die alte Formel in der Wissensbasis durch die aktuelle ersetzt.
- **Abbrechen:** Beim Abbruch werden Änderungen der angezeigten Formel nicht übernommen und der Formel Editor schließt sich.

5.5 Beenden

Nach einem Klick auf Beenden, öffnet sich eine Sicherheitsabfrage (siehe Abbildung 5.18). Erst wenn hier mit Ja bestätigt wird, wird das Programm endgültig beendet. Alle bis dahin nicht gespeicherten Daten gehen verloren.

Abbildung 5.18: Sicherheitsabfrage beim Beenden der Anwendung

Kapitel 6

Klassendiagramme der Kleingruppenarbeit

Die Klassendiagramme zeigen eine UML-Übersicht über die geleistete Programmierarbeit. Diese Arbeit zerfällt in vier Teilbereiche:

1. **Verarbeitung** des Wissens
2. **Markierung**, d.h. Umgang mit den Labeln
3. **Parser**, d.h. das Einlesen der Wissensbasis und die Umsetzung in Formeln
4. **Die Gui** des Programms

Dies wird durch folgende Übersicht dargestellt:

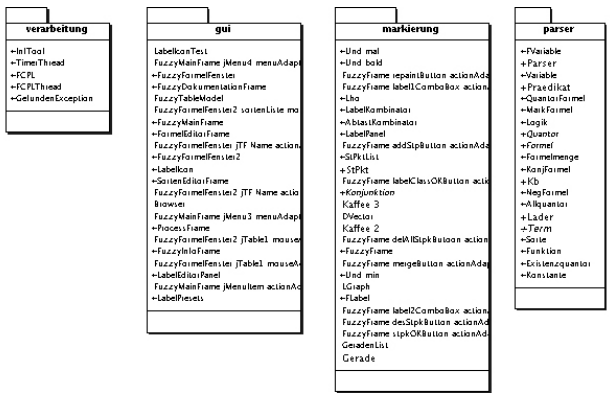


Abbildung 6.1: Übersicht über die einzelnen Kleingruppen

6.1 Verarbeitung

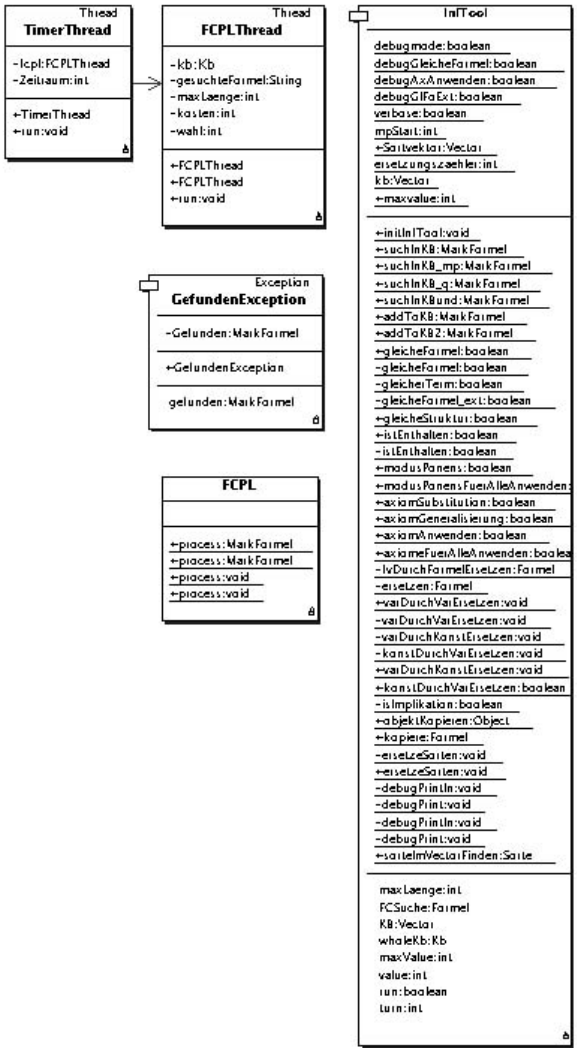


Abbildung 6.2: Die Klassen der Verarbeitung

6.2 Markierung

Für die bessere Übersicht wurde das Klassendiagramm in einzelne Ausschnitte zerlegt:

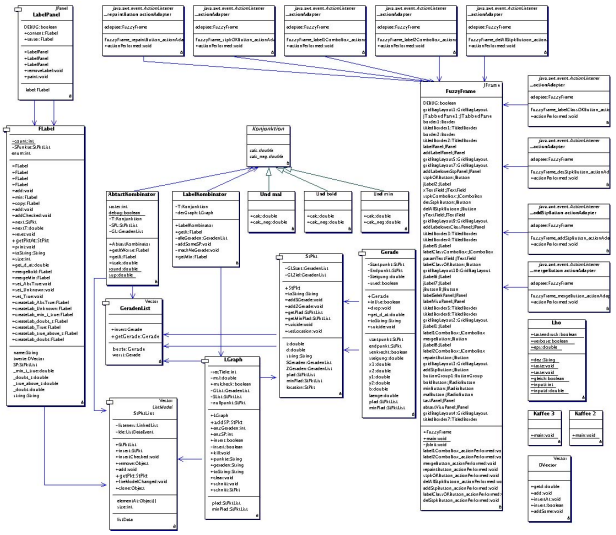


Abbildung 6.3: Klassen für die Labels

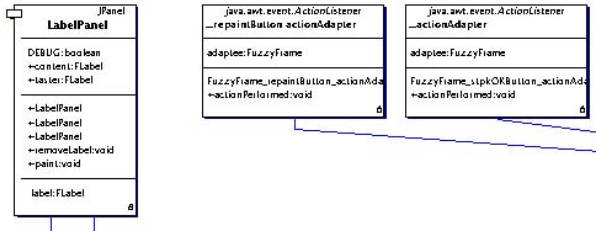


Abbildung 6.4:

Abbildung 6.5:

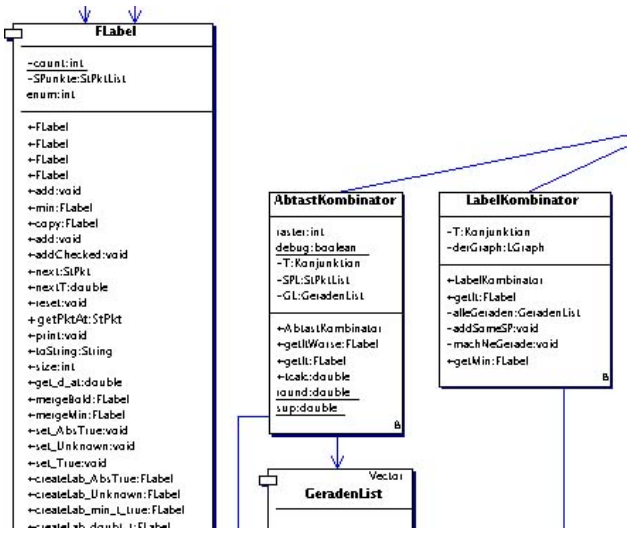


Abbildung 6.6:

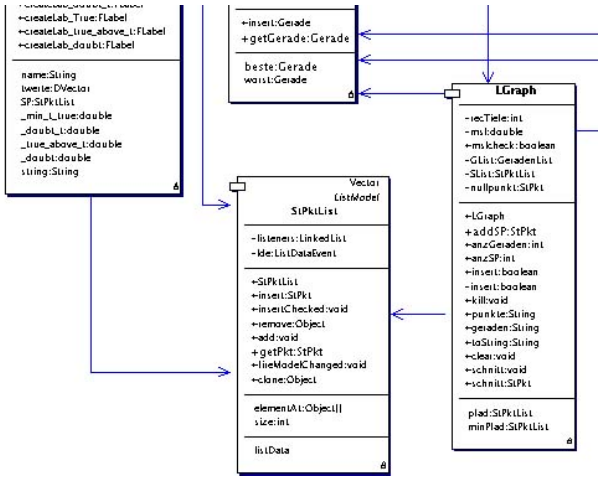


Abbildung 6.7:

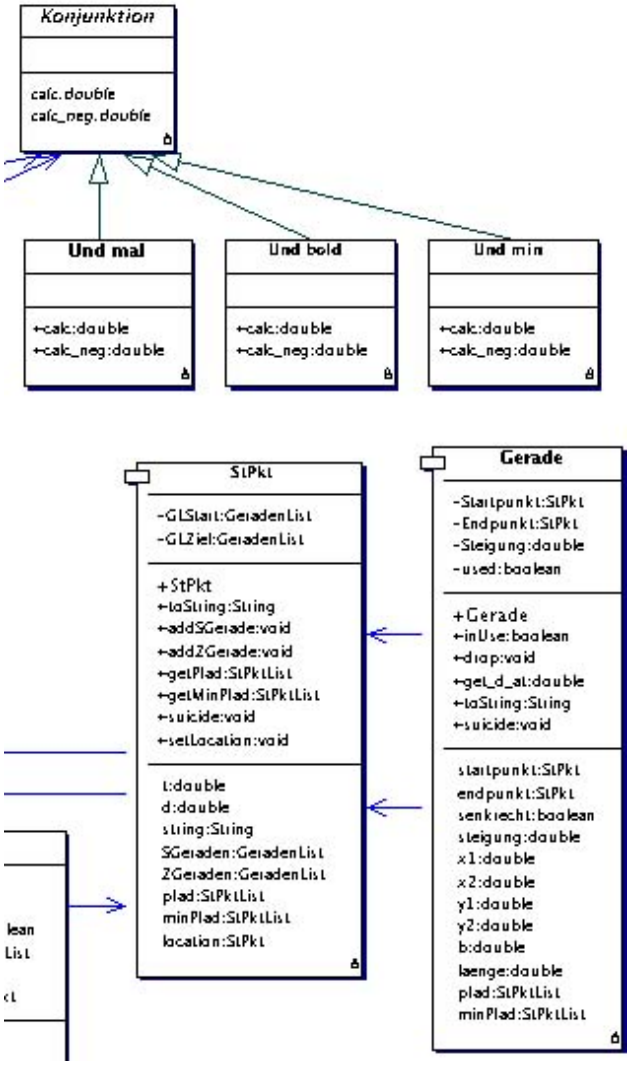


Abbildung 6.8:

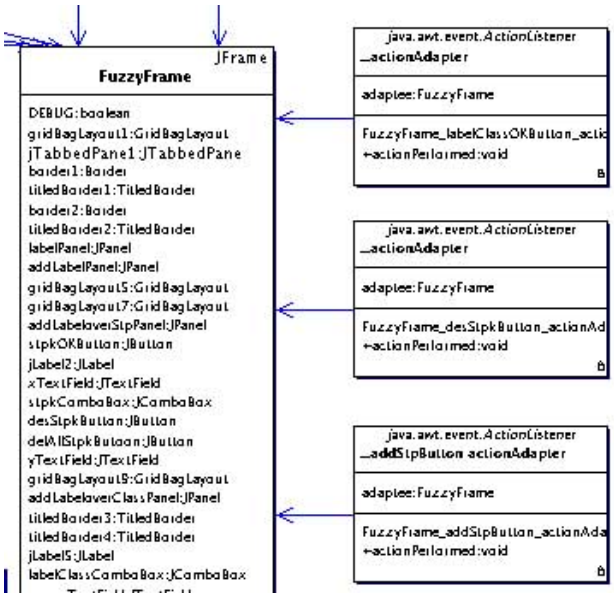


Abbildung 6.9:

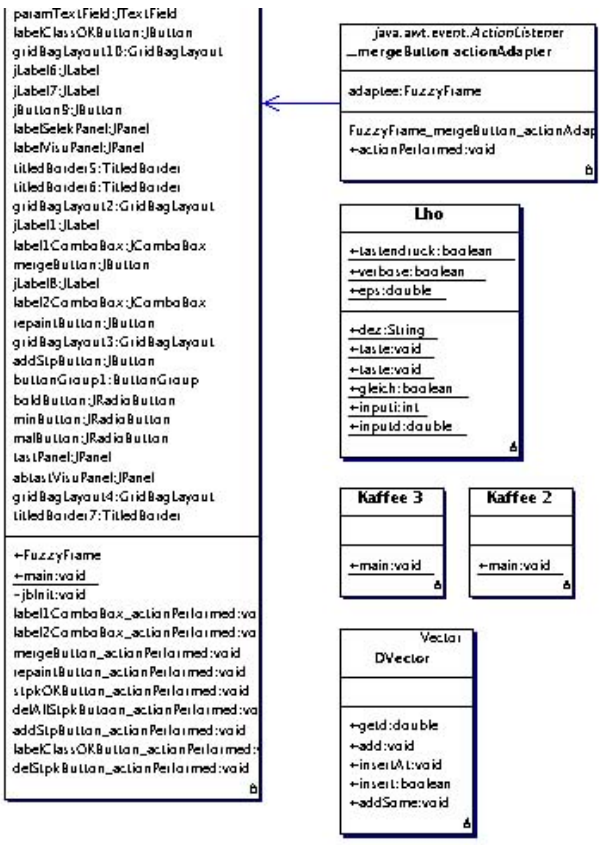


Abbildung 6.10:

6.3 Parser

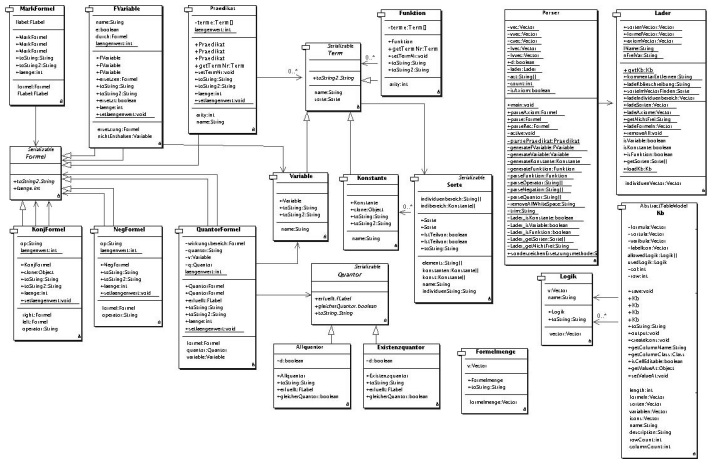


Abbildung 6.11:

Auch dieses Diagramm wurde in einzelne zerlegt:

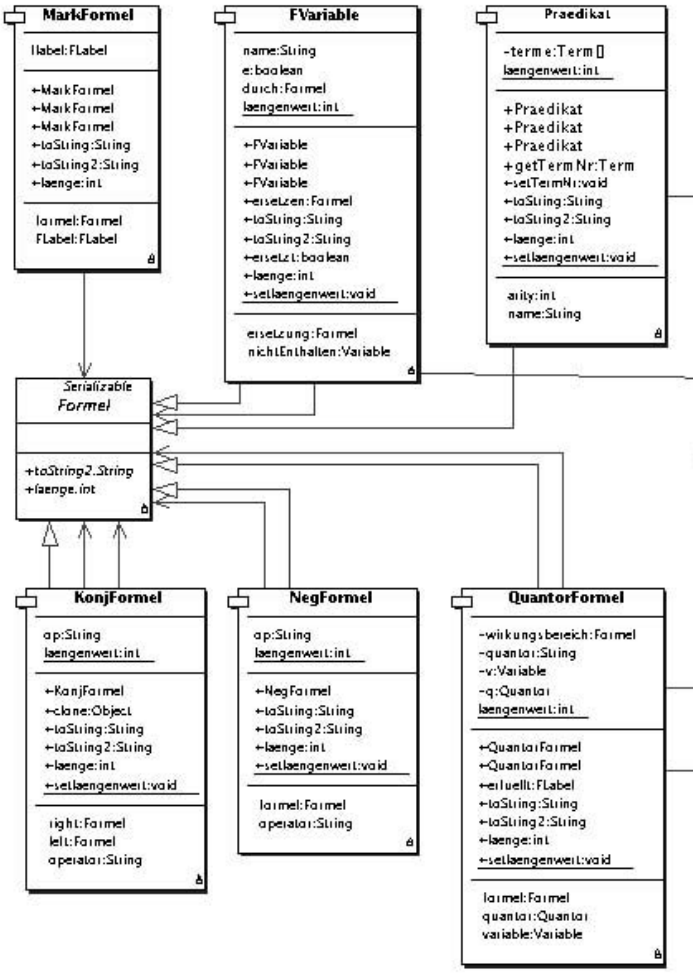


Abbildung 6.12:

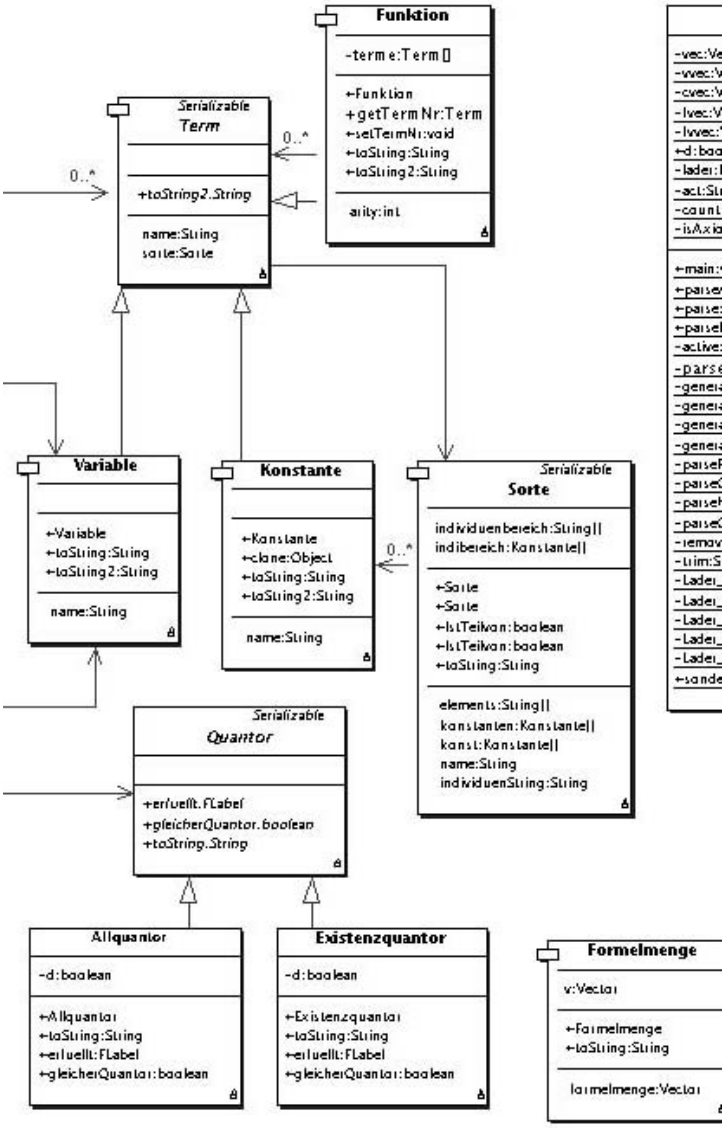


Abbildung 6.13:

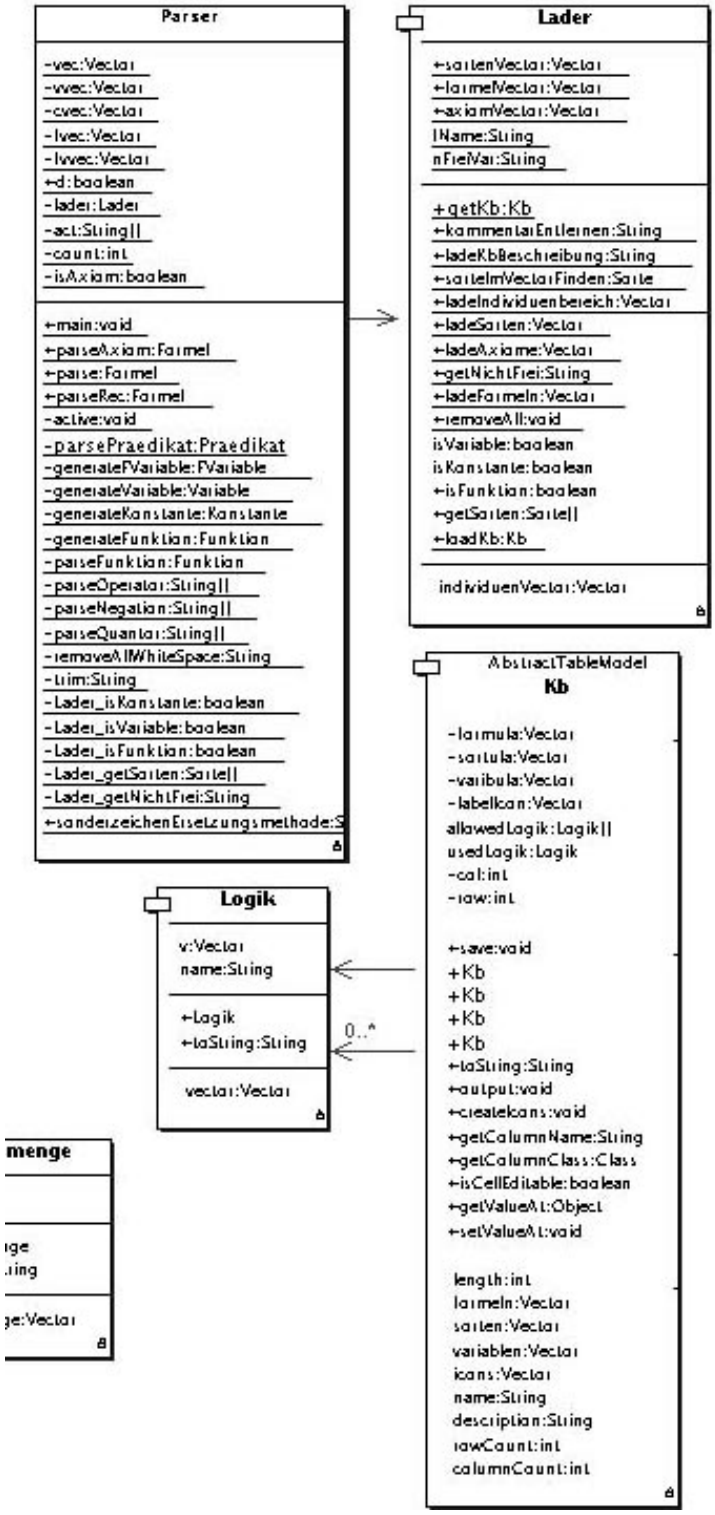


Abbildung 6.14:

6.4 Gui

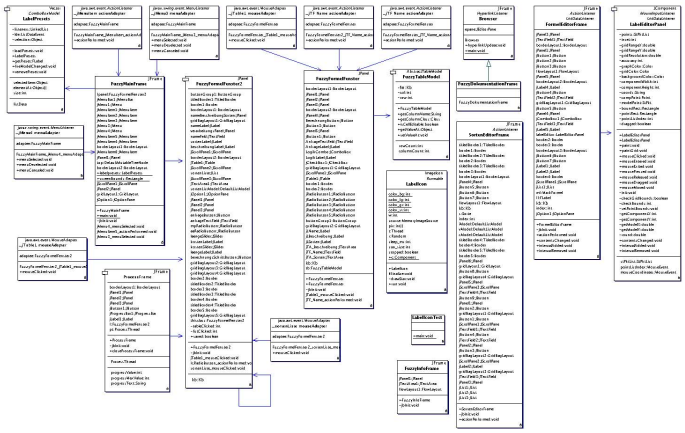


Abbildung 6.15:

Wegen der Übersichtlichkeit erfolgt erneut eine Zerlegung:

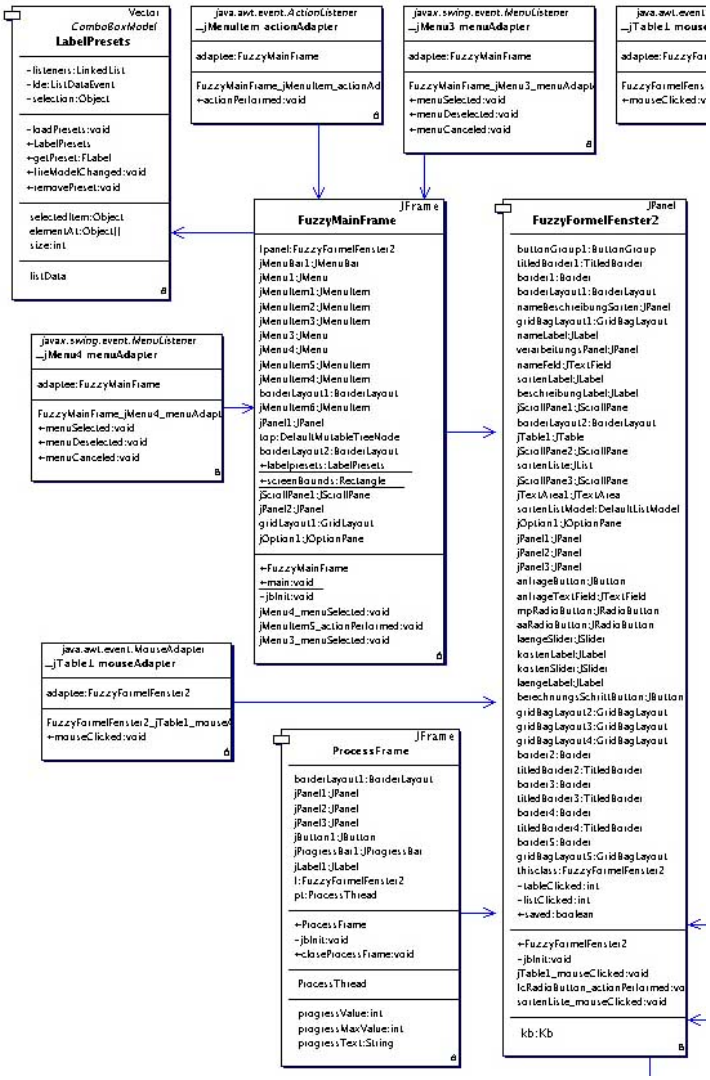


Abbildung 6.16:

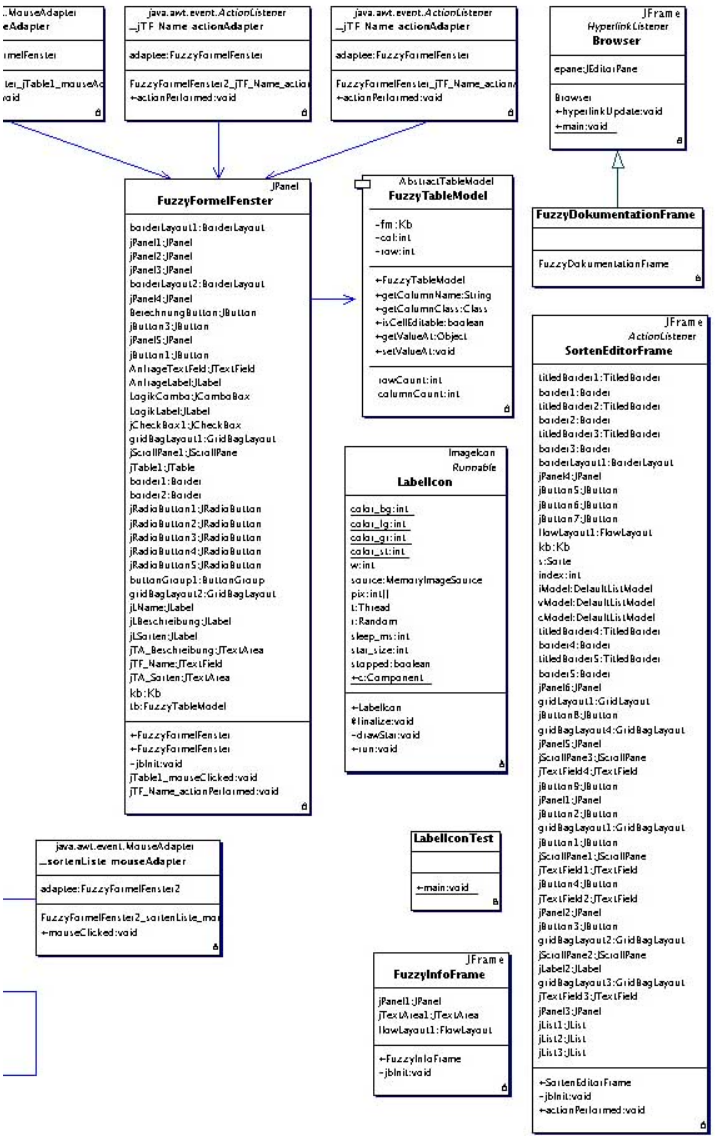


Abbildung 6.17:

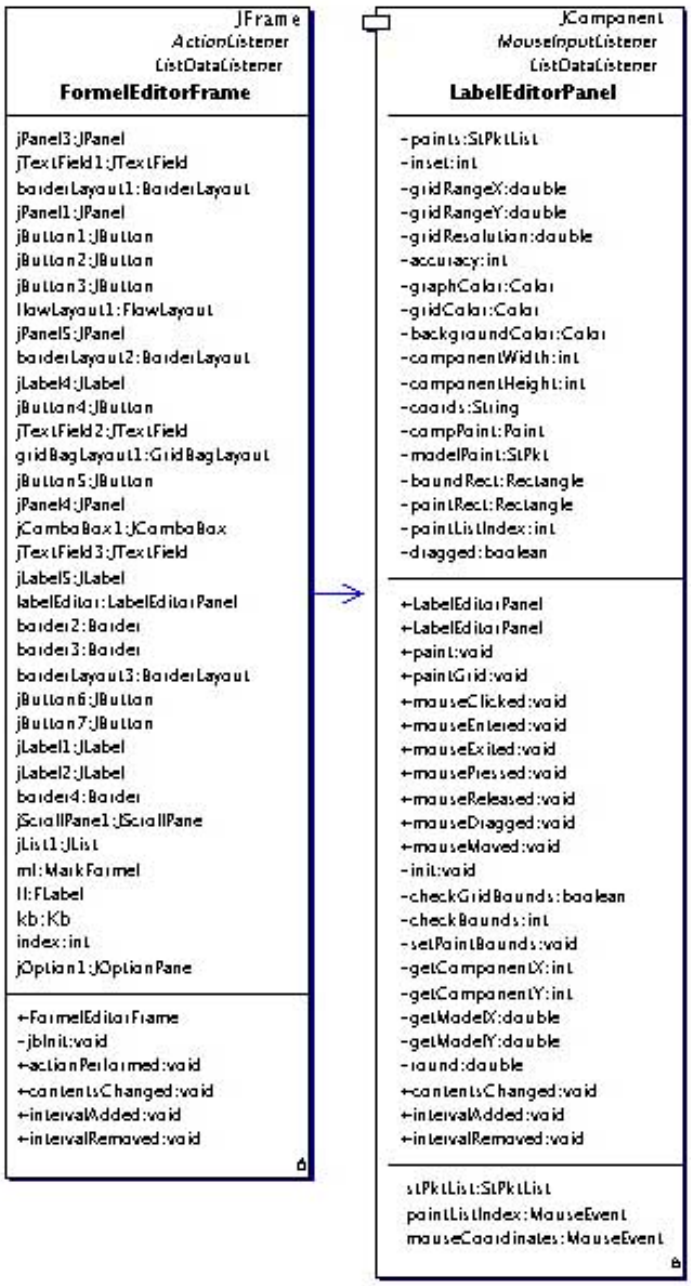


Abbildung 6.18:

Kapitel 7

Exemplarischer Auszug des Quellcodes

Im Folgenden werden wichtige Teile des Quellcodes exemplarisch hervorgehoben und erklärt. Diese Teile richten sich wieder nach der Arbeit der Kleingruppen, wobei sich die Erläuterungen wahlweise auf den echte Quellcode oder Sourcecode stützten.

7.1 Erläuterung des Parser und assoziierter Klassen

7.1.1 Der Parser

Der Parser aus dem gleichnamigen Package macht aus der String-Repräsentation von prädikatenlogischen Formeln und Axiomen Formelbäume. Diese Formelbäume benutzen Formelklassen, die sich ebenfalls in dem Package parser befinden.
Der Parser geht folgendermaßen vor:

1. Entfernen aller Leerzeichen in dem zu parsenden String.
2. Wenn möglich, Aufsplitten am $\rightarrow$-Operator.
3. Wenn möglich, Aufsplitten am $\vee$-Operator.
4. Wenn möglich, Aufsplitten am $\wedge$-Operator.
5. Wenn möglich, Entfernen überflüssiger Klammern („Trimmen“ der Formel).
6. Wenn möglich, Parsen der Negation $\neg$.
7. Wenn möglich, Parsen eines Existenzquantors $\exists$.
8. Wenn möglich, Parsen eines Allquantors $\forall$.
9. Jetzt sind wir an einem Blatt und parsen ein Prädikat bzw. eine Formelvariable.

Die in den einzelnen Schritten erhaltenen Teilformeln werden rekursiv geparst. Beim Parsen von Formeln werden in den Schritten 7-9 in Interaktion mit dem Lader die Sorten der Terme gesetzt. Werden Axiome konvertiert, so werden auf die gleiche Weise nicht-freie Variablen in den Axiomen gekennzeichnet.

Gleichnamige Konstanten, gleichnamige Variablen, gleichnamige Funktionen, gleichnamige Prädikate und gleichnamige Formelvariablen innerhalb einer Formel bzw. eines Axioms sind gleiche Objekte. Dies spart Speicherplatz und ist vorteilhaft für die Formelverarbeitung. Aus diesem Grund werden für Prädikate, Variablen, Konstanten, Funktionen und Formelvariablen jeweils dynamische Arrays von der Klasse java.util.Vector verwaltet.
Der Parser enthält einen Debug-Modus, der u.a. den erstellten Formelbaum in einer Graphensprache ausgibt. Auf diese Weise kann mit dem Paket graphviz (www.research.att.com/sw/tools/graphviz/) z.B. eine Postscriptdatei des Formelbaums erzeugt werden. Im Parser befindet sich außerdem eine GUI-Hilfsmethode für Umwandlung der math. Zeichen in Unicode. Es ist zu beachten, dass Sorten von Prädikaten implizit durch die Terme in den Formeln gesetzt werden. Aus Zeitgründen wurde kein Syntaxcheck implementiert.

7.1.2 Formeldarstellung

Zur Darstellung von prädikatenlogischen Formeln werden die Klassen Formel, Funktion, KonjFormel, Konstante, NegFormel, Praedikat, Variable, QuantorFormel, sowie die Klassen Sorte, Term verwendet.
Sie setzen die rekursive Struktur von prädikatenlogischen Formeln direkt um (siehe auch das Klassendiagramm):

Die abstrakte Klasse Formel hat zum Beispiel die Unterklassen KonjFormel, Praedikat, NegFormel und QuantorFormel, da eine Formel entweder aus einer Verknüpfung von Formeln, einem Prädikat, einer negierten Formel oder einer quantorisierten Formel besteht.
Entsprechend sind zum Beispiel auch die prädikatenlogischen Terme durch die Oberklasse Term, und die Unterklassen Variable, Konstante und Funktion modelliert.
Jeder Term verfügt über eine Sorte, welche durch die Klasse Sorte dargestellt wird.
Die Klasse MarkFormel repräsentiert eine Formel, die durch ein Label (Klasse FLabel) markiert ist und die Klasse kb stellt eine Wissensbasis, also im Prinzip eine Menge von Formeln, da.

7.1.3 LabelIcon

Die Klasse LabelIcon aus dem GUI-Package stellt Formel-Labels grafisch als Unterklasse von javax.swing.ImageIcon dar. Die Klasse ImageIcon implementiert das Interface javax.swing.Icon, so dass eine optimale Integration in die GUI gewährleistet ist. Das FLabel wird in ein Integer-Array gezeichnet, in dem die Farbe der Pixel gespeichert wird. Dann wird mit Hilfe von java.awt.image.MemoryImageSource ein java.awt.Image erzeugt und als Eigenschaft des ImageIcons mittels setImage(...) gesetzt:

```
public LabelIcon(FLabel l, int w, int star_size) { //w: Größe des Icons in
Pixeln
        ...
        source = new MemoryImageSource(w, h, pix, 0, w);
        Image img = c.createImage(source);
        ...
        setImage(img);
}
```

Mit dem Parameter star_size lässt sich die Größe der animierten Sternchen in den Icons einstellen. Dieses Feature ist nicht besonders funktional, sondern dient rein der Ästhetik. Aus diesem Grund sollte bei prozessorlastiger Inferenz dieses Feature abgeschaltet werden.

7.1.4 FuzzyDokumentationFrame

Die Klasse FuzzyDokumentationFrame ist eine Unterklasse von „Browser“. Sie instanziiert die Browser-Klasse mit der URL „file:Handbuch/Handbuch.html“. Der Browser selbst benutzt einen javax.swing.JEditorPane, mit dem text/plain, text/html und text/rtf dargestellt werden kann. Durch die Implementierung eines
javax.swing.event.HyperlinkListener werden Mausklicks auf Hyperlinks registriert und weiterverarbeitet. Der Hilfebrowser unterstützt sogar HTML-Frames und HTTP.

7.2 GUI

Da der Bereich der GUI (*G*raphical *U*ser *I*nterface) auf algorithmischer Ebene eher uninteressant ist, sei hier lediglich eine Beschreibung der Klassen inklusive ihrer wichtigsten Methoden bzw. in ihnen enthaltenen Ideen aufgeführt.

7.2.1 FuzzyMainFrame

Dies ist die Klasse, die aufgerufen werden muß um das Programm überhaupt zu starten. Die Menüstruktur besteht aus einer *jMenuBar*, die 3 Menüpunkte (*jMenu*) bestehen wiederum aus *jMenuItem*s.
Dies ist die „Schaltzentrale“. Hier entscheidet man, ob eine Wissensbasis geladen, gespeichert oder erzeugt werden soll.

Im Punkt „Hilfe“ erhält man Hilfe oder Dokumentation und mit „Beenden“ wird das Programm nach einer weiteren Kontrollabfrage beendet. Dies geschieht mit `System.exit(0)`, damit eine vollständige Garbage-Collection durchgeführt wird.

7.2.2 FuzzyFormelFenster2

Die Klasse `FuzzyFormelFenster2` wird aufgerufen, wenn man *Wissenbasis/Neu* bzw. *Wissenbasis/Laden* ausgewählt hat. Hier wird alles gesteuert, was zur Erstellung bzw. Bearbeitung einer Wissensbasis nötig ist. Die Eingabe des Namens der Wissenbasis und der Anfrage erfolgt jeweils durch ein *jTextField*, die Eingabe der Bescheibung durch eine *jTextArea*.
Die Sorten werden in einer *jList* angezeigt.
Ein Klick auf einen Eintrag in der Sortenliste markiert diese, ein weiterer Klick öffnet über ein eigens angelegtes *Listmodel* die Klasse *SortenEditorFrame*.

Ein Klick auf eine Formel hebt diese hervor, ein weiterer macht ein Editieren der Formel oder ihres Labels möglich. Dazu wird *LabelEditorPanel* aufgerufen.

7.2.3 SortenEditorFrame

SortenEditorFrame erweitert *JFrame* und zeigt für die Sorte und die Eingabe der Individuen, Variablen, Konstanten und Funktionen je ein *jTextField*. Die Listen der vorhandenen Individuen, Variablen, Konstanten und Funktionen sind jeweils *jList*en.
Damit die jeweiligen Listen geordneten Inhalt haben, muß man herausbekommen, in welche Kategorie ein Objekt fällt. Dazu wird folgender Algorithmus angewendet:

```
// es werden alle Objekte aus der Wissenbasis
 durchlaufen
for (int i = 0; i < kb.getVariablen().size();
 i++) {
  Object o = kb.getVariablen().get(i);

  ...
```

```
// wenn o eine Variable ist, dann ...
if (o instanceof parser.Variable) {
  //... frage, ob o auch zur Sorte s gehört.
   Falls ja, dann ...
  if ( ( (Variable) o).getSorte().getName()
  .equals(s.getName()))
    //... zeige den Namen von o in der Liste
    der Variablen der Sorte an.
    vModel.addElement(((Variable) o).getName());
}
...
```

Das ganze wird analog natürlich auch für Funktionen, Individuen und Konstanten durchgeführt.

Hat man einen neuen Wert bei den Individuen, Variablen, Konstanten oder Funktionen Konstanten eingegeben und klickt letztendlich auf „Fertig“, ist nicht bekannt, welche Werte schon da standen und welche neu dazugekommen sind. Aus diesem Grunde werden alle Objekte aus der vorhandenen Wissenbasis mit denen aus der entsprechenden Liste (*DefaultListModel*) nach Namen verglichen und aus der Wissensbasis entfernt, aber sofort danach mit der eventuellen neuen Sorte wieder eingefügt (Beispiel für die Variablen, analog für die anderen):

```
for (int i = 0; i < vModel.size(); i++) {
  for (int k = 0; k < kb.getVariablen().
  size(); k++){
    Object o = kb.getVariablen().get(k);
    if (o instanceof parser.Variable) {
      if ( ( (Variable) o).getName().
      equals(((String)vModel.get(i))))
        kb.getVariablen().remove(k);
    }
  }
  kb.getVariablen().add(new Variable((
  (String)vModel.get(i)), s));
}
```

Dadurch werden Konflikte bezüglich Sortenunterschiede vermieden.

Für alle Objektnamen aus der Wissensbasis wird abschließend getestet, ob in den Funktionen Parameter vorkommen, die bisher nicht definiert wurden und eine entsprechende Fehlermeldung ausgegeben. Wird eine neue Sorte eingegeben, ist der „Fertig“–Button deaktiviert!

7.3 LabelEditorPanel

Auch *LabelEditorPanel* erweitert *JComponent*.

Kapitel 8

Soll-Ist-Vergleich

Der Soll-Ist-Vergleich stellt die geplanten und tatsächlich erreichten Ziele der PG-Arbeit in tabellarischer Form gegenüber. Dabei wird von der im Zwischenbericht erarbeiteten Spezifikation ausgegangen. Er dient zum einen einer zusammengefassten Darstellung der Funtionalität des PG-Produktes und zeigt auf, wo noch Entwicklungsbedarf besteht (für eine eventuelle Weiterentwicklung). Zum anderen schafft er auch die Möglichkeit, den Erfolg der PG-Arbeit anhand der Anzahll der erreichten Ziele zu messen.

8.1 Tabellarischer Soll-Ist-Vergleich

In den folgenden Tabellen bedeuten + grüne Rechecke, daß die jeweilige Funktion implementiert wurde, o gelbe Rechtecke, daß sie nicht wie in der Spezifikation angegeben oder nur teilweise implementiert wurde, und - rote Rechtecke, daß sie nicht implementiert wurde.

8.1.1 Soll-Ist-Vergleich Pre-Processing

Soll	***Ist***
Eingabe einer Wissensbasis	
Eingabe über ein Textfenster	+
Eingabe der Operatoren mittels GUI Knöpfe	-
Ersetzung der Operatoren und Symbole durch Standardsymbole	+
Eingabe einer Signatur für die Wissensbasis	-
Angabe von Sorten für Funktions- und Relationssymbole, Prädikate und Konstanten	o
Eingabe einer textuellen Beschreibung	+
Syntaktische Überprüfung	
Überprüfung auf syntaktische Korrektheit nach Eingabe oder Laden	+

Überprüfung auf anwendbare Logiken und Vermerkung in der Wissensbasis	-
Speichern einer Wissensbasis	
Speichern in einer Text-Datei	+
Speichern der Signatur	-
Speichern der Labels	+
Speichern der anwendbaren Logiken	-
Speichern der Beschreibung	+
Speichern des Verarbeitungmechanismus	-
Laden einer Wissensbasis	+
Editieren einer Wissensbasis	+
Anhängen einer Wissensbasis	-
Labeleditor	
Graphische Anzeige des Labels	+
Textuelle Eingabe der Labels	+
Graphische Eingabe	+
Nur korrekte Labels können eingegeben werden	o

8.1.2 Soll-Ist-Vergleich Processing

Soll	***Ist***
Anfrage	
Eingabe einer Anfrage	+
Eingabe von mehreren Fakten	-
Erstellung einer „Hitliste“ bei leerer Anfrage	-
Logik	
Axiome der Logiken liegen in Textdatei vor	+
Auswahl verschiedener Logiken	-
Wahl der Labelberechnung	-
Verarbeitungsmechanismen	
Forward Chaining	+
Backward Chaining	-
Resolution	-
Abbruch der Verarbeitung	+

Debug Modus	-

8.1.3 Soll-Ist-Vergleich Post-Processing

Soll	***Ist***
Vereinfachung der Wissensbasis	o
Ausgabe der Lösung	
Anzeigen der Lösung nach Verarbeitung	-
Angabe der Verarbeitungstiefe	-
Angabe, ob die Lösung optimal ist	-
Anzeige der veränderten Wissensbasis	+
Interpretation der Lösung	-
Anzeige des Lösungsweges durch Baumstruktur	-
Speichern der veränderten Wissensbasis	
Speichern unter anderem Namen	+
Änderung der Beschreibung	+
Speicherung der verwendeten Logik	-
Weitere Anfrage	+
Neue Verarbeitung starten	+

8.2 Was wir daraus lernen

Wenn wir die komplett erreichten Ziele mit 1, die nur teilweise erreichten mit 0,5 und die nicht erreichten Ziele mit 0 bewerten und dies auf die Gesamtanzahl unserer gesteckten Ziele beziehen, läßt sich die Erfolgsquote der PG-Arbeit zu

$$\frac{22,5}{44} \quad \hat{=} \quad 51\%$$

berechnen.

Dies ist zwar eine recht naive Form der Berechnung des Erfolges unserer Arbeit, doch zeigt sie die Tendenz dahin, daß wir die Komplexität und den Aufwand zu gering eingeschätzt haben und die Schwierigkeiten, die sich aus der verteilten Bearbeitung einer Aufgabenstellung ergeben, nicht vermeiden konnten.

Kapitel 9

Anhang: Der Quellcode der Projektgruppe

Nachstehend finden sich die wichtigsten Klassen des Projektes gruppiert nach den einzelnen Untergruppen, in denen sie bearbeitet wurden.

9.1 Quellcode der Verarbeitung

```
package verarbeitung;

import java.util.*;
import markierung.*;
import parser.*;
import java.io.*;
import gui.*;


public class InfTool {

    static boolean debugmode = false;
    static boolean debugGleicheFormel = false;
    static boolean debugAxAnwenden = false;
    static boolean debugGlFoExt = false;

    static boolean verbose = false;

    static int mpStart = 0;

    public static Vector Sortvektor = new Vector();

    static int ersetzungszaehler;
```

```
    static Vector kb;
    static Kb wholeKb;

/**
 * Maximale Länge, bis zu der Formeln auf Axiome angewendet werden.
 * Defaultwert ist Länge = 8.
 */
static int maxLaenge = 8;

/**
 * Formel nach der gesucht werden soll.
 * Standardmäßig wird nach keiner Formel gesucht (null)
 */
static Formel FCSuche = null;

    // für ProcessFrame
    public static int value, maxvalue;
    public static boolean run;
    public static int turn;

// fuer Heuristiken

/**
 * Hier kann die Länge für Formeln eingestellt werden, auf die noch Axiome
 * angewendet werden.
 */
public static void setMaxLaenge (int i) {
maxLaenge = i;
}

/**
 * Mit dieser Methode kann die Formel, nach der gesucht werden soll
  gesetzt werden.
 * @param f Formel nach der gesucht werden soll
 */
    public static void setFCSuche(Formel f) {
FCSuche=f;
}

/**
 * Diese Methode gibt die Formel nach der gesucht wird zurück
 * (wird z.Zt. nicht benötigt, da jetzt mit der GefundenExceptions
 gearbeitet wird)
 * @see GefundenException
```

```
 * @return Formel, nach der gesucht wird
 */
public static Formel getFCSuche() {
return FCSuche;
}

    // KB - Operationen ----------------------------------------

  /**
 * Hier wird die Wissensbasis gesetzt.
 * @param wissensbasis Die Wissensbasis als Vector
 */
    public static void setKB(Vector wissensbasis) {
        kb = wissensbasis;
    }

/**
 * Hier kann man sich die Wissensbasis übergeben lassen
 * @return Die Wissensbasis
 */
    public static Vector getKB() {
        return kb;
    }

    public static void setWholeKb(Kb kb) {
        wholeKb = kb;
    }

    public static Kb getWholeKb() {
        return wholeKb;
    }

    public static void setMaxValue(int m) {
       maxvalue = m;
    }

    public static void setValue(int v) {
        value = v;
    }

    public static void setRun(boolean b) {
        run = b;
    }
```

```
public static void setTurn(int v) {
    turn = v;
}

public static void initInfTool() {
   value = 0;
   maxvalue = 1;
   run = true;
   turn = 1;
}


/**
 * @return gibt die gesuchte Formel mit
 Markierung zurück,
 falls in der Wissensbasis vorhanden
 (sonst null).
 * @param f Formel, nach der in der Wissensbasis
 gesucht wird.
 */
public static MarkFormel suchInKB(Formel f) {
    debugPrintln("* Suche nach Formel: "+f,
    debugGleicheFormel);
    for (int i = 0; i < kb.size(); i++) {
        MarkFormel kbf = (MarkFormel) kb.get(i);
        if (gleicheFormel(kbf.getFormel(), f,
        new
        Vector(),
        new Vector(), new Vector(), new Vector())) {
            debugPrintln("* Treffer!",
            debugGleicheFormel);
            return kbf;
        }
        else debugPrintln("* nope.",debugGleicheFormel);
    }
    return null;
}

/**
 * Diese Methode wird bei der Anwendung des
 Modus Ponens
 verwendet. Wenn die Implikation, die übergeben
  wird, erfüllt
 werden kann -
```

```
     * das heisst, die Prämisse in der Wissensbasis
     vorhanden ist
     - wird die Prämisse zurückgegeben.
     * Formelvariablen werden berücksichtigt - d.h.,
      falls nötig,
     wird durch Ersetzungen von Formelvariablen
     die übergebene Formel
      mit einer
     * Formel aus der Wissensbasis unifiziert.
     * Die Wissensbasis wird durchlaufen, bis eine
      Unifikation
     möglich ist. Damit weitere Unifikationen möglich
     sind, wird bei
     erneutem Aufruf
     * ab dieser Stelle weitergesucht.
     * @return gibt die Prämisse der übergebenen Formel
      mit Markierung
     zurück, falls die Prämisse in der Wissensbasis
     vorhanden (sonst null).
     * @param fCopy Implikation, deren Prämisse
     gesucht wird.
    */
    public static MarkFormel suchInKB_mp(KonjFormel
     fCopy) throws
    GefundenException{
        debugPrintln("* Suche nach Formel: "+fCopy,
        debugGleicheFormel);
        for (; mpStart < kb.size(); mpStart++) {
            MarkFormel kbf = (MarkFormel) kb.get
            (mpStart);
    Formel kbfCopy=kopiere(kbf.getFormel());
    Formel fLinks=fCopy.getLeft();
    Vector kbfVec=new Vector();
    Vector fVec=new Vector();
            if (gleicheFormel_ext(kbfCopy, fLinks,
            new Vector(), new Vector(), new Vector(),
            new Vector(),
     kbfVec, fVec)) {
if (!kbfVec.isEmpty()) {
for (int j=0; j<kbfVec.size(); j++) {
Formel a = (Formel) kbfVec.get(j);
Formel b = (Formel) fVec.get(j);

if (a instanceof FVariable) kbfCopy =
```

```
fvDurchFormelErsetzen(kbfCopy, (FVariable)a,
 b);
else fCopy = (KonjFormel) fvDurchFormelErsetzen
(fCopy,
(FVariable)b, a);
}
fLinks=fCopy.getLeft();
debugPrintln("FV ersetzt! ",debugGlFoExt);
debugPrintln("1. Formel: " + kbf.getFormel().
toString(),
debugGlFoExt);
debugPrintln("1. Formel danach: " +
 kbfCopy.toString(),
debugGlFoExt);
debugPrintln("2. Formel danach : "
+fLinks.toString(),
debugGlFoExt);
debugPrintln("2. Formel gesamt: "+fCopy.toString(),
debugGlFoExt);
if (!gleicheFormel(kbfCopy, fLinks)) {
debugPrintln("...und Formeln UNgleich

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
continue;
}

else {
debugPrintln("...und Formeln gleich.",
debugGlFoExt);
mpStart++;
return addToKB(kbfCopy, kbf.getFLabel());
}
}
                debugPrintln("* Treffer!",
                debugGleicheFormel);
mpStart=-1;
                return kbf;
            }
            else debugPrintln("* nope.",
            debugGleicheFormel);
        }
mpStart=-1;
        return null;
    }
```

```
    /*
    public static MarkFormel suchInKB_ext(Formel f)
    throws
    GefundenException {
        debugPrintln("* Suche nach Formel: "+
        f,debugGleicheFormel);
        for (int i = 0; i < kb.size(); i++) {
            MarkFormel kbf = (MarkFormel) kb.get(i);
    Formel kbfCopy=kopiere(kbf.getFormel());
    Formel fCopy=kopiere(f);
    Vector kbfVec=new Vector();
    Vector fVec=new Vector();
            if (gleicheFormel_ext(kbfCopy, fCopy,
            new Vector(),
             new Vector(), new Vector(), new Vector(),
     kbfVec, fVec)) {
if (!kbfVec.isEmpty()) {
for (int j=0; j<kbfVec.size(); j++) {
Formel a = (Formel) kbfVec.get(j);
Formel b = (Formel) fVec.get(j);

if (a instanceof FVariable) kbfCopy =
fvDurchFormelErsetzen
(kbfCopy, (FVariable)a, b);
else fCopy = fvDurchFormelErsetzen(fCopy,
(FVariable)b, a);
}
debugPrintln("FV ersetzt! ",debugGlFoExt);
debugPrintln("1. Formel: " + kbf.getFormel().
toString(),
debugGlFoExt);
debugPrintln("1. Formel danach: " + kbfCopy.
toString(),
debugGlFoExt);
debugPrintln("2. Formel danach : "
+fCopy.toString(),
debugGlFoExt);
if (!gleicheFormel(kbfCopy, fCopy)) {
debugPrintln("...und Formeln UNgleich
 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 ",true);
continue;
```

```
}

else {
debugPrintln("...und Formeln gleich.",
debugGlFoExt);
return addToKB(kbfCopy, kbf.getFLabel());
}
}
                debugPrintln("* Treffer!",
                debugGleicheFormel);
                return kbf;
            }
            else debugPrintln("* nope.",
            debugGleicheFormel);
        }
        return null;
    }
    */

    /**
     * Diese Methode wird bei der Anwendung
     von Quantoren
     verwendet - während der Mosus Ponens
     Prozedur. Unifikation
      siehe suchInKB_mp
       @param f Formel, die in der Wissensbasis
        gesucht wird.
       Es ist die Prämisse der Implikation,
       deren Quantor durch
        Einsetzen aufgelöst wurde.
       @param gesFormel Implikation, für die
       der Modus Ponens
       angewandt wird.
     * @return gibt die gesuchte Formel (f) mit
      Markierung zurück,
     falls in der Wissensbasis vorhanden.
     * @see MarkFormel suchInKB_mp(KonjFormel
      fCopy)
    */
    public static MarkFormel suchInKB_q(Formel f,
     Formel
     gesFormel) throws GefundenException {
        debugPrintln("* Suche nach Formel: "
        +f,debugGleicheFormel);
```

```
        for (int i = 0; i < kb.size(); i++) {
            MarkFormel kbf = (MarkFormel) kb.get(i);
    Formel kbfCopy=kopiere(kbf.getFormel());
    Formel fCopy=kopiere(f);
    Vector kbfVec=new Vector();
    Vector fVec=new Vector();
            if (gleicheFormel_ext(kbfCopy, fCopy,
             new Vector(),
             new Vector(), new Vector(), new Vector(),
     kbfVec, fVec)) {
if (!kbfVec.isEmpty()) {
for (int j=0; j<kbfVec.size(); j++) {
Formel a = (Formel) kbfVec.get(j);
Formel b = (Formel) fVec.get(j);

if (a instanceof FVariable) {
kbfCopy = fvDurchFormelErsetzen(kbfCopy,
 (FVariable)a, b);
fvDurchFormelErsetzen(gesFormel,
(FVariable)a, b);
}
else {
fCopy = fvDurchFormelErsetzen(fCopy,
(FVariable)b, a);
fvDurchFormelErsetzen(gesFormel,
(FVariable)b, a);
}
}
debugPrintln("FV ersetzt! ",debugGlFoExt);
debugPrintln("1. Formel: " + kbf.getFormel
().toString(),
debugGlFoExt);
debugPrintln("1. Formel danach: " +
kbfCopy.toString()
,debugGlFoExt);
debugPrintln("2. Formel danach : "
+fCopy.toString(),
debugGlFoExt);
if (!gleicheFormel(kbfCopy, fCopy)) {
debugPrintln("...und Formeln UNgleich
 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 ",true);
continue;
}
```

```
else {
debugPrintln("...und Formeln gleich.",
debugGlFoExt);
return addToKB(kbfCopy, kbf.getFLabel());
}
}
                debugPrintln("* Treffer!",
                debugGleicheFormel);
                return kbf;
            }
            else debugPrintln("* nope.",
            debugGleicheFormel);
        }
        return null;
    }


    /**
     * Falls f eine undverknüpfte Formel ist,
      sucht diese Methode
      in der Wissensbasis getrennt nach dem
      linken
     * und dem rechten Teil der Konjunktion.
     Sind beide Teile
     vorhanden, wird die Konjunktion als neue
     Formel der Wissensbasis hinzugefügt und
     zurückgegeben. Die
      Markierung der neuen Formel ergibt sich
       aus der
     * Bold-Konjunktion der Teilformeln.
     * @param f Konjunktion, deren Teile in
     der Wissensbasis
     gesucht werden.
     * @return zusammengefügte Konjunktion mit
      Markierung
    */
    public static MarkFormel suchInKBund(Formel
    f) throws
    GefundenException{

        if (! (f instanceof KonjFormel)) return null;

        KonjFormel fk = (KonjFormel) f;
```

```
        if (!fk.getOperator().equals("&")) return null;
        MarkFormel links = suchInKB(fk.getLeft());

        if (links == null) return null;
        MarkFormel rechts = suchInKB(fk.getRight());

        if (rechts == null) return null;
        FLabel neuL = links.getFLabel().mergeBold(rechts.
        getFLabel());
        KonjFormel undF = new KonjFormel(links.getFormel(),
        rechts.getFormel(), "&");
        return addToKB(undF, neuL);
    }

    /**
     *  Fügt eine Formel mit Markierung in die
     Wissensbasis ein.
      Ist diese Formel bereits vorhanden, wird
       das
     *  Label aktualisiert, indem aus dem alten
      Label und dem neu übergeben Label das "Minimumlabel"
     gebildet wird,
     *  d.h. es findet eine Oder-verknüpfung statt.
     *  Während der Suche nach einer Anfrage
     wird die Formel mit
      der gesuchten Formel verglichen,
     *  bei Übereinstimmung wird eine
     GefundenException geworfen.
     *  @param f Formel (ohne Markierung),
     die in die Wissensbasis
      eingefügt werden soll
     *  @param l FLabel, mit dem die Formel
      markiert werden soll
     *  @return die markierte Formel, die entweder
      in der Wissensbasis
      gefunden oder neu erzeugt wurde.
     */
    public static MarkFormel addToKB(Formel f, FLabel
     l) throws GefundenException {
        // erzeugte Formel bereits vorhanden in KB?
        debugPrintln("---Hinzufügen von "+f.
        toString2()+" ... ");
MarkFormel alteF = suchInKB(f);
```

```
        if (alteF != null) {
            //ja => Änderung des Labels
            //debugPrintln("---Formel bereits
            vorhanden.");
            alteF.setFLabel(alteF.getFLabel().min(l));

            /* Aktualisierung des Icons??? */


            return alteF;
        }
        else {
            //nein => Hizufügen der neuen Formel.
            debugPrintln("Formel neu ("+ (kb.size()+1)
             + "): "+f.toString());
            MarkFormel neueF = new MarkFormel(f, l);
            kb.add(neueF);
            wholeKb.getIcons().add(new LabelIcon(neueF
            .getFLabel(), 20, 0));
            wholeKb.fireTableDataChanged();

if (FCSuche == null)
             return neueF;
if (gleicheFormel(FCSuche, neueF.getFormel()))
throw new GefundenException(neueF);
else
return neueF;
        }
    }


   /**
 * Fügt eine Formel in die Wissensbasis ein.
 Siehe addToKB.
  Hier wird jedoch keine GefundenException geworfen,
 * diese Methode kann also bei der manuellen
 Eingabe benutzt werden oder wenn keine
 Suchanfrage vorliegt.
 *  @param f Formel (ohne Markierung), die in die
  Wissensbasis eingefügt werden soll
     *  @param l FLabel, mit dem die Formel markiert
      werden soll
     *  @return die markierte Formel, die entweder
```

```
     in der Wissensbasis gefunden oder neu erzeugt wurde.
 *  @see MarkFormel addToKB(Formel f, FLabel l)
*/
public static MarkFormel addToKB2(Formel f,
FLabel l) {
try {
return addToKB(f,l);
}
catch (Exception rtl) {};
return null;
}
```

```
    // Formelvergleich ------------------------------

    /**
     * Testet zwei Formeln auf Gleichheit. Zwei Formeln
     sind gleich, wenn sie durch Umbenennung der
     * Variablen unifizierbar sind.
     * @param a 1. Formel
     * @param b 2. Formel
     * @return true falls Formeln gleich sind, sonst false.
    */
    public static boolean gleicheFormel(Formel a, Formel b)
    {
     return gleicheFormel(a, b, new Vector(), new Vector(),
      new Vector(), new Vector());
    }

    private static boolean gleicheFormel(Formel a, Formel b,
    Vector v1, Vector v2, Vector vfv1, Vector vfv2) {
        debugPrintln("gleiche Formel: "+a+" vs. "+b,
        debugGleicheFormel);
        if ( (a instanceof KonjFormel) && (b
        instanceof KonjFormel)) {
            KonjFormel ak = (KonjFormel) a;
            KonjFormel bk = (KonjFormel) b;
            if (!ak.getOperator().equals(bk.getOperator())) {
                return false;
            }
            return gleicheFormel(ak.getLeft(), bk.getLeft(),
```

```
        v1, v2, vfv1, vfv2) &&
            gleicheFormel(ak.getRight(), bk.getRight(),
             v1, v2, vfv1, vfv2);
    }
    else if ( (a instanceof NegFormel) && (b
    instanceof NegFormel)) {
        NegFormel an = (NegFormel) a;
        NegFormel bn = (NegFormel) b;
        if (!an.getOperator().equals(bn.getOperator())) {
            return false;
        }
        return gleicheFormel(an.getFormel(), bn.getFormel(),
        v1, v2, vfv1, vfv2);
    }
    else if ( (a instanceof QuantorFormel) && (b instanceof
    QuantorFormel)) {
        debugPrint("Quantorformel. ",debugGleicheFormel);
        QuantorFormel aq = (QuantorFormel) a;
        QuantorFormel bq = (QuantorFormel) b;
        if (!aq.getQuantor().gleicherQuantor(bq.getQuantor()))
        {
            debugPrintln("Quantor stimmt nicht überein",
            debugGleicheFormel);
            return false;
        }
        Variable a_var = aq.getVariable();
        Variable b_var = bq.getVariable();
        //Sorte muss gleich sein...
        if (a_var.getSorte() != b_var.getSorte()) {
            debugPrintln("Sorte stimmt nicht überein",
            debugGleicheFormel);
            return false;
        }
        //wenn beide Variablen noch nicht im Vector stehen,
         ist der Index von beiden -1
        if ( (v1.indexOf(a_var)) != (v2.indexOf(b_var))) {
            debugPrintln("Variablenindex stimmt nicht
             überein",debugGleicheFormel);
            return false;
        }
        if (!v1.contains(a_var)) { //-> auch b_var steht
         noch nicht in v2
            v1.add(a_var);
            v2.add(b_var);
```

```
            }
            return gleicheFormel(aq.getFormel(), bq.getFormel()
            , v1, v2, vfv1, vfv2);
        }
        else if ( (a instanceof Praedikat) && (b
        instanceof Praedikat)) {
            Praedikat ap = (Praedikat) a;
            Praedikat bp = (Praedikat) b;
            if (!ap.getName().equals(bp.getName())) {
                return false;
            }
            //!!! Zugriff auf Term[0] mit getTermNr(1) !!!
            //daher Zählung ab 1
            for (int i = 1; i <= ap.getArity(); i++) {
                Term aTerm = ap.getTermNr(i);
                Term bTerm = bp.getTermNr(i);
if (!gleicherTerm(aTerm, bTerm, v1, v2)) return false;

            }
            return true;
        }
else if ((a instanceof FVariable) && (b instanceof
 FVariable)) {
if (vfv1.indexOf(a) != vfv2.indexOf(b)) return false;
else {
if (!vfv1.contains(a)) {
vfv1.add(a);
vfv2.add(b);
}
return true;
}
}
        else return false;
    }

    private static boolean gleicherTerm(Term aTerm, Term
     bTerm, Vector v1, Vector v2) {
     if (aTerm.getSorte()!=bTerm.getSorte()) return false;
     if (aTerm instanceof Variable) {
if (! (bTerm instanceof Variable)) {
return false;
}
if ( (v1.indexOf(aTerm)) != (v2.indexOf(bTerm))) {
return false;
```

```
}
if (!v1.contains(aTerm)) {
v1.add(aTerm);
v2.add(bTerm);
}
}
if (aTerm instanceof Konstante) {
//Konstanten sollten nur in einer Instanz
 existieren
//(sonst: Stringvergleich)
//bTerm keine Konstante->es wird ebenfalls false
zurückgegeben.
if (! aTerm.getName().equals(bTerm.getName() )) {
return false;
}
}

if (aTerm instanceof Funktion) {
if (!(bTerm instanceof Funktion)) return false;
Funktion aFunc=(Funktion)aTerm;
Funktion bFunc=(Funktion)bTerm;
if (aFunc.getArity()!=bFunc.getArity()) return
false;
for(int i=0; i<aFunc.getArity(); i++) {
if (!gleicherTerm(aFunc.getTermNr(i+1),
bFunc.getTermNr(i+1), v1, v2)) return false;
}
}
     return true;
    }


private static boolean gleicheFormel_ext(Formel a,
Formel b, Vector v1, Vector v2, Vector vfv1, Vector vfv2,
 Vector aVec, Vector bVec) {
        debugPrintln("gleiche Formel: "+a+" vs. "+b
        ,debugGleicheFormel);
        if ( (a instanceof KonjFormel) && (b instanceof
         KonjFormel)) {
            KonjFormel ak = (KonjFormel) a;
            KonjFormel bk = (KonjFormel) b;
            if (!ak.getOperator().equals(bk.getOperator())) {
                return false;
            }
```

```
        return gleicheFormel_ext(ak.getLeft(), bk.getLeft(),
         v1, v2, vfv1, vfv2, aVec, bVec) &&
            gleicheFormel_ext(ak.getRight(), bk.getRight(),
             v1, v2, vfv1, vfv2, aVec, bVec);
    }
    else if ( (a instanceof NegFormel) && (b instanceof
     NegFormel)) {
        NegFormel an = (NegFormel) a;
        NegFormel bn = (NegFormel) b;
        if (!an.getOperator().equals(bn.getOperator())) {
            return false;
        }
        return gleicheFormel_ext(an.getFormel(),
        bn.getFormel(), v1, v2, vfv1, vfv2, aVec, bVec);
    }
    else if ( (a instanceof QuantorFormel) && (
    b instanceof QuantorFormel)) {
        debugPrint("Quantorformel. ",debugGleicheFormel);
        QuantorFormel aq = (QuantorFormel) a;
        QuantorFormel bq = (QuantorFormel) b;
        if (!aq.getQuantor().gleicherQuantor(bq.getQuantor())) {
            debugPrintln("Quantor stimmt nicht überein"
            ,debugGleicheFormel);
            return false;
        }
        Variable a_var = aq.getVariable();
        Variable b_var = bq.getVariable();
        //Sorte muss gleich sein...
        if (a_var.getSorte() != b_var.getSorte()) {
            debugPrintln("Sorte stimmt nicht überein",

            debugGleicheFormel);
            return false;
        }
        //wenn beide Variablen noch nicht im Vector
        stehen, ist der Index von beiden -1
        if ( (v1.indexOf(a_var)) != (v2.indexOf(b_var))) {
            debugPrintln("Variablenindex stimmt nicht
             überein",debugGleicheFormel);
            return false;
        }
        if (!v1.contains(a_var)) { //-> auch b_var steht
        noch nicht in v2
            v1.add(a_var);
```

```
                v2.add(b_var);
            }
            return gleicheFormel_ext(aq.getFormel(),
            bq.getFormel(), v1, v2, vfv1, vfv2, aVec, bVec);
        }
        else if ( (a instanceof Praedikat) && (b instanceof
        Praedikat)) {
            Praedikat ap = (Praedikat) a;
            Praedikat bp = (Praedikat) b;
            if (!ap.getName().equals(bp.getName())) {
                return false;
            }
            //!!! Zugriff auf Term[0] mit getTermNr(1) !!!
            //daher Zählung ab 1
            for (int i = 1; i <= ap.getArity(); i++) {
                Term aTerm = ap.getTermNr(i);
                Term bTerm = bp.getTermNr(i);
if (!gleicherTerm(aTerm, bTerm, v1, v2)) return false;

            }
            return true;
        }
else if ((a instanceof FVariable) && (b
instanceof FVariable)) {
if (vfv1.indexOf(a) != vfv2.indexOf(b)) return false;
else {
if (!vfv1.contains(a)) {
vfv1.add(a);
vfv2.add(b);
}
return true;
}
}

else if ((a instanceof FVariable) ^
(b instanceof FVariable)) {
if (aVec.indexOf(a) != bVec.indexOf(b)) return false;
else {
if (!vfv1.contains(a)) {
aVec.add(a);
bVec.add(b);
}
return true;
}
```

```
}
        else return false;
    }


    /**
    * Testet, ob ein Axiom auf eine Formel angewandt
     werden kann. Hierzu muss die Struktur der Prämisse
    * des Axioms mit der Struktur der Formel übereinstimmen.
    * @param ax die Prämisse des Axioms, das angewandt
     werden soll.
    * @param frm die Formel, auf die das Axiom
    angewandt werden soll.
    * @param vec Vektor, der zur Unifizierung der
    Formelvariablen des Axioms mit Formelteilen dient.
     Initialisierung mit einem leeren Vektor.
    * @param cont noch so ein Vektor.
    * @return true falls strukturgleich (Axiom anwendbar),
    sonst false.
    */
    public static boolean gleicheStruktur(Formel ax,
    Formel frm, Vector vec, Vector cont) {
        if ( (ax instanceof KonjFormel) && (frm
        instanceof KonjFormel)) {
         KonjFormel ak = (KonjFormel) ax;
         KonjFormel bk = (KonjFormel) frm;
         if (!ak.getOperator().equals(bk.getOperator
         ())) {
             return false;
         }
         return gleicheStruktur(ak.getLeft(),
         bk.getLeft(), vec, cont)
                && gleicheStruktur(ak.getRight(),
                 bk.getRight(), vec, cont);
        }
        else if ( (ax instanceof NegFormel) &&
        (frm instanceof NegFormel)) {
            NegFormel an = (NegFormel) ax;
            NegFormel bn = (NegFormel) frm;
            if (!an.getOperator().equals(bn.
            getOperator())) {
                return false;
            }
            return gleicheStruktur(an.getFormel(),
```

```
            bn.getFormel(), vec, cont);
        }
        else if ( (ax instanceof QuantorFormel) &&
         (frm instanceof QuantorFormel)) {
            QuantorFormel aq = (QuantorFormel) ax;
            QuantorFormel bq = (QuantorFormel) frm;
            if (!aq.getQuantor().gleicherQuantor
            (bq.getQuantor())) {
                return false;
            }

            return gleicheStruktur(aq.getFormel(),
             bq.getFormel(), vec, cont);
        }
        else if (ax instanceof FVariable) {
            //cont ist Kontrollvektor, zum Test der
            Strukturgleichheit erforderlich.
            if (vec.indexOf(frm) != cont.indexOf(ax))
            return false;
            FVariable fv = (FVariable) ax;
            Variable av = fv.getNichtEnthalten();
            if ( (av != null) && (istEnthalten(frm,av)))
             return false;
            //wenn frm nicht in vec, dann ax nicht
            in cont. Werden mit gleichem
            //Index hinzugefügt.
            if (!vec.contains(frm)) {
                vec.add(frm);
                cont.add(ax);
            }
        return true;
        }
        else return false;
    }

    /**
     * Testet, ob Variable v in Formel a enthalten ist.
     * @param a die Formel
     * @param v die Variable
     * @return true falls enthalten, sonst false.
    */
    public static boolean istEnthalten(Formel a,
     Variable v) {
        if (a instanceof Praedikat) {
```

```
            Praedikat ap=(Praedikat)a;
            for (int i = 1; i <= ap.getArity(); i++) {
Term aTerm = ap.getTermNr(i);
if (istEnthalten(aTerm, v)) return true;
            }
            return false;
}
else if (a instanceof FVariable) {
            return false;
}
else if (a instanceof KonjFormel) {
            KonjFormel ak = (KonjFormel) a;
            return istEnthalten(ak.getLeft(),v) ||
            istEnthalten(ak.getRight(),v);
}
else if (a instanceof NegFormel) {
            NegFormel an = (NegFormel) a;
            return istEnthalten(an.getFormel(),v);
}
else if (a instanceof QuantorFormel) {
            QuantorFormel aq = (QuantorFormel) a;
            return istEnthalten(aq.getFormel(),v);
}
else return false;
    }


    private static boolean istEnthalten(Term aTerm,
     Variable v) {
     if (aTerm == v) {
              return true;
}
if (aTerm instanceof Funktion) {
Funktion aFunc=(Funktion)aTerm;
for(int i=0; i<aFunc.getArity(); i++) {
if (istEnthalten(aFunc.getTermNr(i+1),v)) return true;
}
}
return false;
    }
```

```
    //Modus Ponens --------------------------------

    /**
     * Versucht, den Modus Ponens auf eine Formel
      anzuwenden. Hierzu wird zunächst die Prämisse
       in der Wissensbasis
     * gesucht. Wird sie gefunden (bzw. eine Formel,
     die mit der Prämisse unifiziert werden kann),
      kann der Modus
     * Ponens angewandt werden: die Konklusion wird
     als neue Formel zur Wissensbasis hinzugefügt.
     Die Markierung
     * ergibt sich aus der bold-Konjunktion der
     Ursprungsformel, auf die der Modus-Ponens angewandt
      wird, und der
     * Prämisse, die in der Wissensbasis gefunden wurde.
     * @param mf markierte Formel (sollte Implikation sein),
      auf die der Modus Ponens angewandt wird.
     * @return true, falls der Modus Ponens erfolgreich
     angewandt werden konnte, sonst false.
     * @see MarkFormel suchInKB_mp(KonjFormel fCopy)
    */
    public static boolean modusPonens(MarkFormel mf)
    throws GefundenException{
        //Implikation? sonst -> exit
boolean back=false;
        if (! (mf.getFormel()instanceof KonjFormel)) {
            return false;
        }
        KonjFormel f = (KonjFormel) mf.getFormel();
        if (!isImplikation(f)) {
            return false;
        }

        //Praemisse in KB ?
KonjFormel fcopy=(KonjFormel) kopiere(f);
mpStart=0;
while(mpStart!=-1) {
         MarkFormel mPraemisse = suchInKB_mp(fcopy);
         if (mPraemisse == null) {
                //oder, falls linke Seite undverknüpft,
                 einzelne Elemente in KB ?
            mPraemisse = suchInKBund(f.getLeft());
            //oder, falls linke Seite QuantorFormel,
```

```
             ist der Quantor erfüllt?
             if (f.getLeft() instanceof QuantorFormel) {
                 QuantorFormel qf=(QuantorFormel)
                 f.getLeft();
                 FLabel nLabel=qf.erfuellt(fcopy, kb);
                 if (nLabel==null) return back;
                else {
                     mPraemisse=addToKB(qf, nLabel);
                 }
             }
          //nein -> exit
          }
          if (mPraemisse == null) {
             return back;
          }
          //ja-> super, passendes Label erzeugen.
//System.out.println("Mergen von "+mf.getFLabel()+
" und "+mPraemisse.getFLabel());
          FLabel neuesLabel = mf.getFLabel().
          mergeBold(mPraemisse.getFLabel());
//System.out.println("-> "+neuesLabel);
          addToKB(fcopy.getRight(), neuesLabel);
back=true;
}
        return back;
    }

    /**
     * Testet für alle Formeln aus der Wissensbasis,
     ob der Modus Ponens für sie angewendet werden kann.
     * In allen erfolgreichen Versuchen wird die
     dabei entstehende neue Formel zur Wissensbasis hinzugefügt.
     * @see Boolean modusPonens(MarkFormel mf)
     * @return immer true.
     */
public static boolean modusPonensFuerAlleAnwenden()
 throws GefundenException {
          // für ProcessFrame
          maxvalue = kb.size();

for (int zaehler = 0; (zaehler < kb.size()
 && run); zaehler++) {  // 'run' ermöglicht Abbruch

if (verbose) System.out.print("\r teste
```

```
 Formel "+(zaehler+1)+" von "+kb.size()+": ");
                          // für ProcessFrame
                        value = zaehler+1;

modusPonens(((MarkFormel)kb.get(zaehler)));
}
return true;
}

    // Axiome ---------------------------------------

    /**
     * Führt das Substitutionsaxiom aus: Wird
     eine Allquantorformel übergeben, wird eine neue Formel
      zur
     * Wissensbasis hinzugefügt, in der alle Vorkommen
      der durch den Allquantor gebundenen Variablen durch
     * eine Konstante ersetzt werden.
     * @param mf - die markierte Formel, auf die das
     Axiom angewandt wird
     * @param k - die Konstante, durch die
     die Variable ersetzt wird.
     * @return true, falls die Formel eine
     Allquantorformel ist, sonst false
    */
    public static boolean axiomSubstitution
    (MarkFormel mf, Konstante k) throws GefundenException{

if ((mf.getFormel() instanceof QuantorFormel) &&
 (((QuantorFormel)mf.getFormel()).getQuantor()
instanceof Allquantor)) {
QuantorFormel qf = (QuantorFormel)
kopiere(mf.getFormel());
Variable v = qf.getVariable();
varDurchKonstErsetzen(qf.getFormel(), v, k);
addToKB(qf.getFormel(),mf.getFLabel());
return true;
}
else return false;
    }

    /**
     * Führt das Generalisierungssaxiom aus:
     Alle Vorkommen einer Konstanten werden
```

```
      durch eine Variable ersetzt.
     * Diese Variable wird durch einen
     Existenzquantor gebunden. Die so gewonnene
      neue Formel wird zu Wissensbasis
     * hinzugefügt.
     * @param mf die markierte Formel,
     auf die das Axiom angewandt wird
     * @param k die Konstante, die ersetzt
      werden soll.
     * @return immer true.
    */
    public static boolean axiomGeneralisierung
    (MarkFormel mf, Konstante k)
     throws GefundenException{
        Formel a = (Formel) kopiere
        (mf.getFormel());
Variable v = new Variable("var_"+
(new Date()).getTime(),k.getSorte());
if (!konstDurchVarErsetzen(a,v,k)) return false;
wholeKb.getVariablen().add(v);
QuantorFormel qf = new QuantorFormel("?",v,a);
addToKB(qf,mf.getFLabel());
return true;
    }

    /**
    * wendet EIN Axiom auf EINE Formel an.
    Zunächst findet ein Test auf Strukturgleichheit
    statt, dann
    * werden die benötigten Ersetzungen
    durchgeführt.
    * @param ax das Axiom, das angewendet wird.
    * @param frm Formel, auf dies
     das Axiom angewendet wird.
    * @return true, falls Axiom erfolgreich
    angewendet, sonst false.
    * @see InfTool#gleicheStruktur(Formel ax,
    Formel frm, Vector vec, Vector cont).
   */
    public static boolean axiomAnwenden(KonjFormel
    ax, MarkFormel frm) throws GefundenException{
        KonjFormel cloneAxiom;
        Vector vec = new Vector();
        debugPrintln("Das Axiom:  " + ax.toString2(),
```

```
        debugAxAnwenden);
        debugPrintln("Die Formel: " + frm.toString2(),
         debugAxAnwenden);
/*
 if (!gleicheStruktur(ax.getLeft(),
 frm.getFormel(), vec, new Vector())) {
            return false;
        }
*/
        // Klonen des Axioms
Variable qv = null;
if (frm.getFormel() instanceof QuantorFormel) {
QuantorFormel qf = (QuantorFormel) frm.getFormel();
qv = qf.getVariable();
}
        cloneAxiom = (KonjFormel) objektKopieren(ax);
if (qv != null) {
//System.out.println(f);
if (cloneAxiom.getLeft() instanceof QuantorFormel) {
Variable bav=((QuantorFormel)
cloneAxiom.getLeft()).getVariable();
varDurchVarErsetzen(cloneAxiom, bav ,qv);
}
}
        debugPrintln("Das Axiom (geklont):
         " + cloneAxiom.toString2(), debugAxAnwenden);
 if (!gleicheStruktur(cloneAxiom.getLeft(),
 frm.getFormel(), vec, new Vector())) {
            return false;
        }
        ersetzungszaehler=0;
        Formel f =  ((KonjFormel)
        ersetzen(cloneAxiom, vec)).getRight();
addToKB (f,frm.getFLabel());
        return true;
    }

/**
* wendet alle Axiome auf alle Formeln der
Wissensbasis an.
* @param ax Vector, der alle Axiome enthält.
* @return immer true
* @see InfTool#axiomAnwenden(KonjFormel ax,
 MarkFormel frm)
```

```
*/
public static boolean axiomeFuerAlleAnwenden(Vector ax)
throws GefundenException{
//für alle Formeln der WB
int KBGroesse = kb.size();
String s="";
         // für ProcessFrame
            maxvalue = kb.size();

for (int i = 0; (i < KBGroesse && run); i++){
// 'run' ermöglicht Abbruch
            // für ProcessFrame
            value = i + 1;
            if (verbose) s = ("Formel " +
            (i + 1) + " / " + KBGroesse + " - ");
            //werden alle Axiome angewandt
            if ( ( (MarkFormel) kb.get(i)).
            laenge() < maxLaenge) {
             for (int j = 0; j < ax.size(); j++) {
                 if (verbose) System.out.print(
                 "\r " + s + "Axiom " + (j + 1) + " /
                 " + (ax.size() + 2) + ": ");
                    InfTool.axiomAnwenden( (
                    (KonjFormel) (ax.get(j))), (
                    (MarkFormel) (kb.get(i))));
                }
    }
            // hartkodierte Axiome
            Vector iv = Lader.getIndividuenVector();
            if (verbose) System.out.print("\r " + s +
            "Subst. und Gen.: ");
            for (int j = 0; j < iv.size(); j++) {
             Konstante[] konst = ( (Sorte) iv.get(j))
             .getKonst();
                for (int k = 0; k < konst.length; k++) {
                 InfTool.axiomSubstitution( (MarkFormel)

                 kb.get(i), konst[k]);
                    if ( (MarkFormel) kb.get(i)).laenge()
                    < maxLaenge) {
InfTool.axiomGeneralisierung( (MarkFormel)
kb.get(i), konst[k]);
}
                }
```

```
            }
        }
return true;
}

    //Ersetzungen ----------------------------------------

    // in a soll fv durch e ersetzt werden.
    private static Formel fvDurchFormelErsetzen(Formel a,
     FVariable fv, Formel e) {
        if (a == fv) {
            FVariable av = (FVariable) a;
            /*
               Wenn die FVariable, die man erreicht hat vorher
               schon mal ersetzt wurde, dann muß die FVariable
               mit der selben Formel wie vorher ersetzt werden.
               Ansonsten wird im ELSE-Teil die FVariable durch
               die i-te Formel des Vektors ersetzt und die
               FVariable
               entsprechend "markiert", so daß man weiß, daß diese
               FVariable schon mal ersetzt wurde.
            */
            if (av.ersetzt()) a = av.getErsetzung();
            else a = av.ersetzen(e);
        }
else if (a instanceof KonjFormel) {
            KonjFormel ak = (KonjFormel) a;
            ak.setLeft(fvDurchFormelErsetzen
            (ak.getLeft(), fv, e));
            ak.setRight(fvDurchFormelErsetzen
            (ak.getRight(), fv, e));
        }
        else if (a instanceof NegFormel) {
            NegFormel an = (NegFormel) a;
            an.setFormel(fvDurchFormelErsetzen
            (an.getFormel(), fv, e));
        }
        else if (a instanceof QuantorFormel) {
            QuantorFormel aq = (QuantorFormel) a;
            aq.setFormel(fvDurchFormelErsetzen(aq.
            getFormel(), fv, e));
        }
return a;
    }
```

```
/**
 In dieser Methode wird das Ersetzen der
 "Axiomenvariablen"
 durch die Formel durchgeführt.
 Dies passiert mit einem rekursiven Abstieg. Das
 (geklonte) Axiom wird solange rekursiv durchlaufen,
  bis alle
 FVariablen ersetzt wurden.
*/
private static Formel ersetzen(Formel a, Vector v) {
    /*
    Wenn der Formelbaum rekursiv durchlaufen wird,
    kann es sein, daß man zu einem Teil kommt, den
    man schon ersetzt hat, also ein Prädikat erreicht.
    Dann kann man in diesem Teil aufhören
    */
    if (a instanceof Praedikat) {
        return a;
    }

    /*
    Wenn man im Axiom eine FVariable erreicht,
     muß diese
    ersetzt werden.
    */
    if (a instanceof FVariable) {
        FVariable av = (FVariable) a;
        /*
           Wenn die FVariable, die man erreicht hat vorher
           schon mal ersetzt wurde, dann muß die FVariable
           mit der selben Formel wie vorher ersetzt werden.
           Ansonsten wird im ELSE-Teil die FVariable durch
           die i-te Formel des Vektors ersetzt und die
           FVariable
           entsprechend "markiert", so daß man weiß,
           daß diese
           FVariable schon mal ersetzt wurde.
        */
        if (av.ersetzt()) {
            a = av.getErsetzung();
        }
        else {
```

```
                if (ersetzungszaehler < v.size()) {
                    Formel f = (Formel) v.elementAt
                    (ersetzungszaehler++);
                    a = av.ersetzen(f);
                }
            }
        }
        else if (a instanceof KonjFormel) {
            KonjFormel ak = (KonjFormel) a;
            ak.setLeft(ersetzen(ak.getLeft(), v));
            ak.setRight(ersetzen(ak.getRight(), v));
        }
        else if (a instanceof NegFormel) {
            NegFormel an = (NegFormel) a;
            an.setFormel(ersetzen(an.getFormel(), v));
        }
        else if (a instanceof QuantorFormel) {
            QuantorFormel aq = (QuantorFormel) a;
            aq.setFormel(ersetzen(aq.getFormel(), v));
        }
        return a;
    }

/**
 * In der übergebenen Formel wird eine Variable
 durch eine andere ersetzt.
 * Wenn die Variable in der Formel mehrmals v
 orkommt, wird jedes Vorkommen durch
 * die übergebene Varable ersetzt.
 * @param a Formel, in der die Ersetzung von
 Variablen stattfinden soll
 * @param v Variable die ersetzt werden soll
 * @param k Variable durch die ersetzt werden
  soll
 */
public static void varDurchVarErsetzen(Formel a,
 Variable v, Variable k){
        if (a instanceof Praedikat) {
         Praedikat ap=(Praedikat)a;
for (int i = 1; i <= ap.getArity(); i++) {
             Term aTerm = ap.getTermNr(i);
                if (aTerm == v) {
ap.setTermNr(i,k);
                }
```

```
                if (aTerm instanceof Funktion) {
     varDurchVarErsetzen((Funktion)aTerm,
     v, k);
                }

            }
}
        else if (a instanceof FVariable) {
FVariable afv = (FVariable)a;
if (afv.getNichtEnthalten()==v)
afv.setNichtEnthalten(k);
        }
        else if (a instanceof KonjFormel) {
            KonjFormel ak = (KonjFormel) a;
varDurchVarErsetzen(ak.getLeft(),v,k);
varDurchVarErsetzen(ak.getRight(),v,k);
        }
        else if (a instanceof NegFormel) {
            NegFormel an = (NegFormel) a;
varDurchVarErsetzen(an.getFormel(),v,k);
        }
        else if (a instanceof QuantorFormel) {
            QuantorFormel aq = (QuantorFormel) a;
Variable vq=aq.getVariable();
if (vq == v) {
             aq.setVariable(k);
}
varDurchVarErsetzen(aq.getFormel(),v,k);
        }
    }

    private static void varDurchVarErsetzen
    (Funktion aFunc, Variable v, Variable k) {
for(int i=1; i<=aFunc.getArity(); i++) {
Term aTerm = aFunc.getTermNr(i);
            if (aTerm == v) {
aFunc.setTermNr(i,k);
            }
            if (aTerm instanceof Funktion) {
varDurchVarErsetzen((Funktion)aTerm,
 v, k);
}
  }
    }
```

```
    private static void varDurchKonstErsetzen
    (Funktion aFunc, Variable v, Konstante k) {
for(int i=1; i<=aFunc.getArity(); i++) {
Term aTerm = aFunc.getTermNr(i);
            if (aTerm.getName().equals
            (v.getName())) {
aFunc.setTermNr(i,k);
            }
            if (aTerm instanceof Funktion) {
varDurchKonstErsetzen((Funktion)aTerm
 v, k);
}
  }
    }

    private static void konstDurchVarErsetzen
    (Funktion aFunc, Variable v, Konstante k) {
for(int i=1; i<=aFunc.getArity(); i++) {
Term aTerm = aFunc.getTermNr(i);
            if (aTerm.getName().equals
            (k.getName())) {
aFunc.setTermNr(i,v);
            }
            if (aTerm instanceof Funktion) {
konstDurchVarErsetzen((Funktion)aTerm,
 v, k);
}
  }
    }

/**
 * In der übergebenen Formel wird jedes Vorkommen
 der Variable v durch die
 * Konstante k ersetzt.
 * @param a Formel, in der die Ersetzung der
 Variablen durch die Konstante durchgeführt
 werden soll
 * @param v Variable, die ersetzt werden soll
 * @param k Konstante, durch die die Variable
 ersetzt werden soll
 */
    public static void varDurchKonstErsetzen
    (Formel a, Variable v, Konstante k){
```

```
        if (v.getSorte()==k.getSorte()) {
            if (a instanceof Praedikat) {
                Praedikat ap=(Praedikat)a;
for (int i = 1; i <= ap.getArity(); i++) {
                    Term aTerm = ap.getTermNr(i);
                    if (aTerm == v) {
ap.setTermNr(i,k);
                    }
                    if (aTerm instanceof
                    Funktion) {
     varDurchKonstErsetzen((Funktion)
     aTerm,v,k);
                    }
                }
            }
            else if (a instanceof FVariable) {
;
            }
            else if (a instanceof KonjFormel) {
                KonjFormel ak = (KonjFormel) a;
varDurchKonstErsetzen(ak.getLeft(),v,k);
varDurchKonstErsetzen(ak.getRight(),v,k);
            }
            else if (a instanceof NegFormel) {
                NegFormel an = (NegFormel) a;
varDurchKonstErsetzen(an.getFormel(),v,k);
            }
            else if (a instanceof QuantorFormel) {
                QuantorFormel aq = (QuantorFormel) a;
Variable vq=aq.getVariable();
if (vq == v) {
                    System.out.println("Variable
                    doppelt gebunden");
}
else varDurchKonstErsetzen(aq.getFormel()
,v,k);
            }
}
    }

/**
 * In der übergebenen Formel wird jedes
 Vorkommen der Konstante k durch die
 * Variable v ersetzt.
```

```
 * @param a Formel, in der die Ersetzung der
 Konstanten durch die Variable durchgeführt
 werden soll
 * @param v Variable, durch die die Konstante
  ersetzt werden soll
 * @param k Konstente, die ersetzt werden soll
 */
    public static boolean konstDurchVarErsetzen
    (Formel a, Variable v, Konstante k) {
boolean ersetzt=false;
        if (a instanceof Praedikat) {
            Praedikat ap=(Praedikat)a;
            for (int i = 1; i <= ap.getArity();
             i++) {
                Term aTerm = ap.getTermNr(i);
if (aTerm.getName().equals(k.getName())) {
                    ap.setTermNr(i,v);
ersetzt=true;
}
if (aTerm instanceof Funktion) {
konstDurchVarErsetzen((Funktion)aTerm,
v,k);
}
            }
        }
else if (a instanceof FVariable) {
;
}
else if (a instanceof KonjFormel) {
KonjFormel ak = (KonjFormel) a;
konstDurchVarErsetzen(ak.getLeft(),v,k);
konstDurchVarErsetzen(ak.getRight(),v,k);
}
else if (a instanceof NegFormel) {
NegFormel an = (NegFormel) a;
konstDurchVarErsetzen(an.getFormel(),v,k);
}
else if (a instanceof QuantorFormel) {
QuantorFormel aq = (QuantorFormel) a;
konstDurchVarErsetzen(aq.getFormel(),v,k);
}
return ersetzt;
    }
```

```
   //Hilfsoperationen -------------------------

   //testet ob f eine Implikation ist... vielleicht
    sollte diese Methode eher
   //in KonjFormel stehen...
   private static boolean isImplikation
   (KonjFormel f) {
       if (f.getOperator().equals(">")) return true;
       else return false;
   }

   /**
* Methode zum klonen von Objekten.
* Es werden beliebige Objekte (hier sind es meist
 Formeln) geklont bzw. kopiert.
* @see InfTool#kopiere(Formel f)
* @param einObjekt Das Objekt, das geklont
werden soll
* @return das geklonte Objekt
*/
   public static Object objektKopieren(Object
    einObjekt) {
       Object a = null;
       try {
           ByteArrayOutputStream baos = new
           ByteArrayOutputStream();
           ObjectOutputStream oos = new

           ObjectOutputStream(baos);
           oos.writeObject(einObjekt);
           ByteArrayInputStream bais = new
           ByteArrayInputStream(baos.toByteArray());
           ObjectInputStream ois = new
           ObjectInputStream(bais);
           a = ois.readObject();
       }
       catch (Exception e) {
           e.printStackTrace();
       }
       return a;
   }
```

```
/**
 * Diese Methode kopiert bzw.
  klont die übergebene Formel.
 * Dabei wird ALLES geklont, auch die
 Sorten. Da dann die Sorten doppelt
 * vorhanden sind und dies nicht
 gewünscht ist, wird die Methode ersetzeSorten(Formel a)
 * aufgerufen.
 * @see InfTool#ersetzeSorten(Formel
 a)
 * @param f Formel, die geklont bzw.
  kopiert werden soll
 * @return Die geklonte bzw. kopierte Formel
 */
    public static Formel kopiere(Formel f) {
Formel g = (Formel) objektKopieren(f);
ersetzeSorten(g);
return g;
    }

/**
 * Ersetzt die Sorten in Termen.
 * @see InfTool#ersetzeSorten(Formel a)
 * @param aTerm Ein Term in dem die
 Sorten ersetzt werden sollen
 */
    private static void ersetzeSorten
    (Term aTerm) {
    Sorte s=aTerm.getSorte();
aTerm.setSorte(Lader.
sorteImVectorFinden(s.getName()));
if (aTerm instanceof Funktion) {
Funktion aFunc = (Funktion)aTerm;
for(int i=1; i<=aFunc.getArity(); i++) {
Term bTerm = aFunc.getTermNr(i);
ersetzeSorten(bTerm);
}
        }
    }

/**
 * Diese Methode ersetzt bei einer
 kopierten Formel die Sorten.
```

```
 * Beim kopieren einer Formel wird
  praktisch alles geklont, auch die Sorten.
 * Das klonen von Sorten ist aber
 unerwünscht, da natürlich keine neuen Sorten
 * entstehen sollen. Deshalb werden
  in der kopierten bzw. geklonten Formel
 * die (geklonten) Sorten durch die
 entsprechend vorhandenen Sorten
 * ersetzt.
 * @param a Formel, in der die Sorten
  ersetzt werden sollen
 */
    public static void ersetzeSorten(Formel a) {
        if (a instanceof Praedikat) {
            Praedikat ap=(Praedikat)a;
            for (int i = 1; i <= ap.getArity();
             i++) {
                Term aTerm = ap.getTermNr(i);
ersetzeSorten(aTerm);
            }
        }
        else if (a instanceof FVariable) {
            ;
        }
        else if (a instanceof KonjFormel) {
            KonjFormel ak = (KonjFormel) a;
            ersetzeSorten(ak.getLeft());
            ersetzeSorten(ak.getRight());
        }
        else if (a instanceof NegFormel) {
            NegFormel an = (NegFormel) a;
            ersetzeSorten(an.getFormel());
        }
        else if (a instanceof QuantorFormel) {
            QuantorFormel aq = (QuantorFormel) a;
            Variable v=aq.getVariable();
            Sorte s=v.getSorte();
            v.setSorte(Lader.
            sorteImVectorFinden(s.getName()));
            ersetzeSorten(aq.getFormel());
        }
    }

    // Debug und temporäre Hilfs- und Testmethoden
```

```
    ----------------------------

    private static void debugPrintln(Object s) {
        if (debugmode) System.out.println(s);
    }

    private static void debugPrint(Object s) {
        if (debugmode) System.out.print(s);
    }

    private static void debugPrintln(String s,
     boolean b) {
        if (b) System.out.println(s);
    }

    private static void debugPrint(String s,
    boolean b) {
        if (b) System.out.print(s);
    }

    public static Sorte sorteImVectorFinden(
    String sortenName) {
        for (int i = 0; i < Sortvektor.size(); i++) {
            Sorte aktuelleSorte = (Sorte)
            Sortvektor.elementAt(i);
            if (aktuelleSorte.getName().
            equals(sortenName)) {
                return aktuelleSorte;
            }
        }
        return null;
    }
}



package verarbeitung;

import java.util.*;
import java.io.*;
import parser.*;

public class FCPL {
```

```
/**
 * Methode wird z.Zt. von der GUI nicht benutzt
 */
public static MarkFormel process(Kb kb,
Vector ax, Formel f, int maxLaenge, int kosten,
int maxAusfuehrung) {


InfTool.setMaxLaenge(maxLaenge);
FVariable.setlaengenwert(kosten);

InfTool.setFCSuche(f);
        InfTool.setWholeKb(kb);
        InfTool.setKB(kb.getFormeln());

try {
            int i = 0;
            while( i < maxAusfuehrung ) {
             InfTool.
             axiomeFuerAlleAnwenden(ax);
                InfTool.
                modusPonensFuerAlleAnwenden();
                i++;
            }
        } catch (GefundenException g) {
         InfTool.setFCSuche(null);
            return g.getGefunden();
        }
        InfTool.setRun(false);
        System.out.println("\n *************
         \n \n  Formel konnte nicht gefunden
         werden: \n \n ***************");
        InfTool.setFCSuche(null);
        return null;

}

/**
 * Mit dieser Methode wird der Forward-Chaining
  Algo gestartet.
 * Es wird hier noch eine Formel, nach
 der gesucht werden soll übergeben
```

```
 * @param kb die Wissensbasis
 * @param ax die Axiome
 * @param f Formel nach der gesucht
 werden soll
 * @param maxLaenge Maximale Länge, bis
 zu der Formeln auf Axiome angewendet
  werden sollen
 * @param kosten Kosten, die eine
 Formelvariable bei der Längenberechnung haben soll
 */
public static MarkFormel process(Kb kb,
Vector ax, Formel f, int maxLaenge, int kosten) {

InfTool.setMaxLaenge(maxLaenge);
FVariable.setlaengenwert(kosten);

InfTool.setFCSuche(f);
        InfTool.setWholeKb(kb);
        InfTool.setKB(kb.getFormeln());

try {
while( true && InfTool.run) {
InfTool.axiomeFuerAlleAnwenden(ax);
InfTool.modusPonensFuerAlleAnwenden();
                                InfTool.turn++;
}
} catch (GefundenException g) {
         InfTool.setRun(false);
            InfTool.setFCSuche(null);
            System.out.println("Angefragte
            Formel gefunden");
            return g.getGefunden();
}
                return null;
/*
System.out.println("\n *************
\n \n  Formel konnte nicht gefunden
werden: \n \n ***************");
InfTool.setFCSuche(null);
return null;
*/
}

/**
```

```
 * Mit dieser Methode wird der
  Forward-Chaining Algo gestartet.
 * Es wird entsprechend der Wahl
 des Benutzers entweder der Modus
 Pones oder die Axiome angewendet
 * @param wahl Was ausgeführt werden
  soll, entweder Modus Ponens oder Axiome
 * @param kb die Wissensbasis
 * @param ax die Axiome
 * @param maxLaenge Maximale Länge,
 bis zu der Formeln auf Axiome
 angewendet werden sollen
 * @param kosten Kosten, die eine
  Formelvariable bei der
  Längenberechnung haben soll
 */
   public static void process(int wahl,
    Kb kb, Vector ax,
     int maxLaenge, int kosten ) {
     try {
         InfTool.setMaxLaenge(maxLaenge);
            FVariable.setlaengenwert(kosten);
            InfTool.setWholeKb(kb);
            InfTool.setKB(kb.getFormeln());

switch (wahl) {
             case 4:
                 System.out.println("\n----
                  Modus Ponens anwenden ... ----");
                   InfTool.modusPonensFuerAlleAnwenden();
                   break;
               case 5:
                  System.out.println("\n---- Axiome
                   anwenden ... ----");
                  InfTool.axiomeFuerAlleAnwenden(ax);
                  break;

            }
}
        catch (GefundenException g) {
         InfTool.setRun(false);
            g.printStackTrace();
        }
        InfTool.setRun(false);
```

```
}
}
```

```
package verarbeitung;

import parser.*;

public class FCPLThread extends Thread {

    private Kb kb;
private String gesuchteFormel;
private int maxLaenge = 10;
private int kosten = 1;
    private int wahl;

/*
public FCPLThread(Kb kb, Formelmenge
 ax, String gesuchteFormel, int maxLaenge,
  int kosten) {
          this.kb = kb;
          this.gesuchteFormel = gesuchteFormel;
          this.maxLaenge = maxLaenge;
          this.kosten = kosten;
}
*/

/**
 * Konstruktor für einen "Verarbeitungs-Thread".
 * Der gesamte Forward-Chaining Algo wird
  als eigener Thread gestartet. Hier wird
 * zusätzlich noch eine Formel angegeben,
 nach der gesucht werden soll
 * @param kb Die Wissensbasis
 * @param gesuchteFormel Die Formel, die angefragt
 wurde bzw. gesucht werden soll
 * @param maxLänge Maximale Länge, bis zu der
  Formeln auf Axiome angewendet werden
 * @param kosten Kosten der Formelvariablen
 */
    public FCPLThread(Kb kb, String gesuchteFormel,
     int maxLaenge, int kosten) {
```

```
     this.kb = kb;
        this.gesuchteFormel = gesuchteFormel;
        this.maxLaenge = maxLaenge;
        this.kosten = kosten;
        this.wahl = -1;

          /*Für das ProcessFrame */
          InfTool.initInfTool();
    }

/**
 * Konstruktor für einen "Verarbeitungs-Thread".
 * Der gesamte Forward-Chaining Algo wird als
 eigener Thread gestartet.
 * Es wird entsprechend der Wahl des Benutzers
 entweder der Modus Pones oder die Axiome angewendet
 * @param kb Die Wissensbasis
 * @param maxLänge Maximale Länge, bis zu der
 Formeln auf Axiome angewendet werden
 * @param kosten Kosten der Formelvariablen
 * @param wahl Was ausgeführt werden soll
 (entweder Modus Ponens oder Axiome)
 */
    public FCPLThread(Kb kb, int maxLaenge,
    int kosten, int wahl) {
     this.kb = kb;
        this.gesuchteFormel = null;
        this.maxLaenge = maxLaenge;
        this.kosten = kosten;
        this.wahl = wahl;

        /*Für das ProcessFrame */
        InfTool.initInfTool();
    }

/**
 * Run-Methode, die zum starten des
 Verarbeitungsthreads benötigt wird
 */
public void run() {
//System.out.println("test FCPL");
//FCPL.process(fm.getFormelmenge(),ax.
getFormelmenge(),Parser.parse(gesuchteFormel),15,1);
        if (wahl > -1)
```

```
          FCPL.process(wahl, kb, Lader.
          axiomVector, maxLaenge, kosten);
         else
             FCPL.process(kb,Lader.axiomVector,
             Parser.parse(gesuchteFormel),maxLaenge,kosten);
}
}
```

```
package verarbeitung;

import parser.*;

/**
 * Diese Exception wird benutzt, um
 innerhalb eines Verarbeitungsschrittes
 * (axiomeFuerAlleAnwenden() bzw.
 modusPonensFuerAlleAnwenden()) die Verarbeitung
 * abzubrechen, wenn die angefragte
  bzw. gesuchte Formel gefunden wurde.
 * @see InfTool#axiomeFuerAlleAnwenden(Vector ax)
 * @see InfTool#modusPonensFuerAlleAnwenden()
 */
public class GefundenException extends Exception {

private MarkFormel Gefunden;

public GefundenException(MarkFormel mf) {
this.Gefunden = mf;
}

/**
 * Es wird die gefundene mrkierte Formel
 zurückgegeben. Diese muß logischerweise
 * mit der angefragten bzw. gesuchten
 Formel übereinstimmen
 * @return Die gesuchte (und gefundene)
 markierte Formel
 */
public MarkFormel getGefunden() {
```

```
return this.Gefunden;
}
}
```

9.2 Quellcode der Markierung

Dvector.java

```
package markierung;

import java.util.*;
import markierung.*;

/*
 * DVector ist ein erweiterter Vector, der zum
 vereinfachten Verwalten
 * von Doubles dient.
 */

class DVector
    extends Vector {

  //liefert den Double-Wert an Position i
  public double getd(int i) {
    return ( (Double) get(i)).doubleValue();
  }

  //fügt einen Double-Wert d ans Ende des Vector
   ein
  public void add(double d) {
    add(new Double(d));
  }

  //fügt einen Double-Wert d an die Stelle i
  in den Vector ein
  public void insertAt(double d, int i) {
    insertElementAt(new Double(d), i);
  }

  /* fügt einen Double-Wert d sortiert in den Vector ein.
   * liefert false zurück falls der Wert schon in der Liste
```

```
   * existierte (-> kein Einfügen), sonst true
   */
  public boolean insert(double d) {
    int i = 0;
    for (i = 0; i < size(); i++) {
      if (getd(i) == d) {
        return false;
      }
      if (getd(i) > d) {
        break;
      }
    }
    insertAt(d, i);
    return true;
  }

  //fügt einen DVector dv sortiert ein
  public void addSome(DVector dv) {
    for (int i = 0; i < dv.size(); i++) {
      insert(dv.getd(i));
    }
  }
}
```

FLabel.java

```
package markierung;

/*
 * FLabel repräsentiert das Label, das zur Markierung

  von logischen
 * Formeln dient.
 */
public class FLabel {

  //count:=Anzahl erzeugter FLabel
  private static int count = 0;

  private String name;
  private StPktList SPunkte;
  int enum;
```

```
/*
 * Konstruktoren:
 * Erzeugen ein neues FLabel mit den Stützpunkten (0|0)
  und (1|1).
 * Wird eine Stützpunktliste übergeben, werden deren
  Stützpunkte
 * übernommen. Wird ein String übergeben, wird er als
  Bezeichner
 * des Labels übernommen. Ansonsten wird ein
 Standardname erzeugt.
 */
public FLabel() {
  count++;
  name = "neues Label " + count;
  SPunkte = new StPktList();
  SPunkte.insert(new StPkt(0.0, 0.0));
  SPunkte.insert(new StPkt(1.0, 1.0));
  enum = 0;
}

public FLabel(StPktList SL) {
  count++;
  name = "neues Label " + count;
  SPunkte = SL;
  enum = 0;
}

public FLabel(String name) {
  count++;
  this.name = name;
  SPunkte = new StPktList();
  SPunkte.insert(new StPkt(0.0, 0.0));
  SPunkte.insert(new StPkt(1.0, 1.0));
  enum = 0;
}

public FLabel(String name, StPktList SL) {
  count++;
  this.name = name;
  SPunkte = SL;
  enum = 0;
}
```

```
public void setName(String name) {
  this.name = name;
}

public String getName() {
  return name;
}

//erzeugt einen neuen Stützpunkt mit den
übergebenen Koordinaten
//und fügt ihn zum Label hinzu.
public void add(double x, double y) {
  add(new StPkt(x, y));
}

//liefert ein Minimumlabel aus dem Label und
übergebenen Label A
public FLabel min(FLabel A) {
  LabelKombinator LK = new LabelKombinator(new Und_min());
  return LK.getMin(this, A);
}

//erzeugt ein neues FLabel mit den gleichen Stützpunkten.
public FLabel copy() {
  StPktList SPL = (StPktList) SPunkte.clone();
  return new FLabel(SPL);
}

//fügt StPkt S zum Label hinzu.
public void add(StPkt S) {
  SPunkte.insert(S);
}

public void addChecked(StPkt s) {
  SPunkte.insertChecked(s);
}

//next, nextT, reset -> Iterator.
public StPkt next() {
  if (enum >= SPunkte.size()) {
    return null;
  }
  else {
    return (StPkt) SPunkte.elementAt(enum++);
```

```
  }
}

public double nextT() {
  if (enum >= SPunkte.size()) {
    return 1.1;
  }
  else {
    return ( (StPkt) SPunkte.elementAt(enum)).getT();
  }
}

public void reset() {
  enum = 0;
}

//liefert den i-Stützpunkt des Labels
public StPkt getPktAt(int i) {
  return SPunkte.getPkt(i);
}

//erzeugt eine Ausgabe der Stützpunkte des
Labels.
public void print() {
  for (int i = 0; i < SPunkte.size(); i++) {
    StPkt S = SPunkte.getPkt(i);
    System.out.println("StPkt " + i + ": " +
    S.toString());
  }
}

public String toString() {
  return SPunkte.toString();
}

//liefert Anzahl der Stützpunkte
public int size() {
  return SPunkte.size();
}

//liefert den d-Wert zu beliebiger
t-Koordinate (0.0 <= t <= 1.0 ).
public double get_d_at(double t) {
  if ( (t < 0.0) || (t > 1.0)) {
```

```
    return 0.0;
  }
  if (1.0 == t) {
    return 1.0;
  }
  StPkt S = null;
  reset();
  while (nextT() <= t) {
    S = next();
  }
  Gerade gtemp = new Gerade(S, next());
  reset();
  return gtemp.get_d_at(t);
}

//liefert die t-Werte, die bei einer
 Kombination des Labels mit
//einem anderen Label interessant seien könnten...
public DVector getTwerte() {
  DVector V = new DVector();
  StPkt S = null;
  reset();
  while (nextT() <= 1.0) {
    S = next();
    V.insert(S.getT()); //-> t-Werte des Labels
    V.insert(S.getD()); //-> d-Werte des Labels
    V.insert(1.0 - S.getT()); //"inverse" T-Werte
  }
  reset();
  return V;
}

//liefert die Stützpunktliste. Dieser ist
ein Vector.
//Einfügen in diese Liste nur mit insert(StPkt S),
 damit Sortierung
//erhalten bleibt (besser noch: add(StPkt S)
 aus Label verwenden).
//Methode eigentlich überflüssig...
sollte gelöscht werden...
public StPktList getSP() {
  return SPunkte;
}
```

```
//kombiniert das Label mit dem Label
zl mit der Bold-T-Norm und
//liefert das Ergebnis zurück
public FLabel mergeBold(FLabel zl) {
  LabelKombinator LK = new LabelKombinator
  (new Und_bold());
  return LK.getIt(this, zl);
}

//kombiniert das Label mit dem Label zl
 mit der Min-T-Norm und
//liefert das Ergebnis zurück
public FLabel mergeMin(FLabel zl) {
  LabelKombinator LK = new LabelKombinator(
  new Und_min());
  return LK.getIt(this, zl);
}

//die folgenden Methoden fügen Stützpunkte
ein, um auf einfache
//Weise die Standardlabel zu erzeugen.
public void set_AbsTrue() {
  add(new StPkt(1.0, 0.0));
}

public void set_Unknown() {
  add(new StPkt(0.0, 1.0));
}

public void set_min_t_true(double t) {
  add(new StPkt(t, 0.0));
  add(new StPkt(t, 1.0));
}

public void set_doubt_t(double t) {
  add(new StPkt(0.0, t));
  add(new StPkt(t, t));
}

public void set_True() {
  ;
}

public void set_true_above_t(double t) {
```

```
  add(new StPkt(t, 0.0));
  add(new StPkt(t, t));
}

public void set_doubt(double d) {
  add(new StPkt(0.0, d));
  add(new StPkt(1.0, d));
}

//die folgenden Methoden sind überflüssig
und dienen nur
//der Kompatibilität mit der der GUI.
public FLabel createLab_AbsTrue() {
  FLabel temp = new FLabel();
  temp.add(new StPkt(1.0, 0.0));
  return temp;
}

public FLabel createLab_Unknown() {
  FLabel temp = new FLabel();
  temp.add(new StPkt(0.0, 1.0));
  return temp;
}

public FLabel createLab_min_t_true(double t) {
  FLabel temp = new FLabel();
  add(new StPkt(t, 0.0));
  add(new StPkt(t, 1.0));
  return temp;
}

public FLabel createLab_doubt_t(double t) {
  FLabel temp = new FLabel();
  temp.add(new StPkt(0.0, t));
  temp.add(new StPkt(t, t));
  return temp;
}

public FLabel createLab_True() {
  FLabel temp = new FLabel();
  return temp;
}

public FLabel createLab_true_above_t(double t) {
```

```
    FLabel temp = new FLabel();
    temp.add(new StPkt(t, 0.0));
    temp.add(new StPkt(t, t));
    return temp;
  }

  public FLabel createLab_doubt(double d) {
    FLabel temp = new FLabel();
    temp.add(new StPkt(0.0, d));
    temp.add(new StPkt(1.0, d));
    return temp;
  }

  public String getString() {
   String s="";
   for (int i=0; i<SPunkte.size(); i++) {
s=s+SPunkte.getPkt(i).getString()+",";
}

return s.substring(0,s.length()-1);


/*
for (int i = 0; i < SPunkte.size(); i++) {
      StPkt S = SPunkte.getPkt(i);
      System.out.println("StPkt " + i + ": "
       + S.toString());
    }
    */
  }

}
```

Gerade.java

```
package markierung;

/*
 * eine Gerade verbindet zwei Stützpunkte.
 */
class Gerade {
```

```
private StPkt Startpunkt;
private StPkt Endpunkt;
private double Steigung;
private boolean senkrecht;
//used zeigt an, ob die Gerade im Graphen
'interessant' ist
private boolean used;

private double x1, y1, x2, y2;

/*
 * Eine Gerade wird durch das Übergeben
 zweier Stützpunkte erzeugt.
 * Dabei wird sofort die Steigung ermittelt.
 * Diese Punkte sollten verschieden sein und
  die Steigung sollte
 * positiv sein. Ansonsten wird der Vorgang
 nicht abgebrochen,
 * jedoch eine Warnung ausgegeben, da im
  weiteren Programmverlauf
 * Probleme auftreten könnten.
 */
public Gerade(StPkt S, StPkt E) {
  Startpunkt = S;
  Endpunkt = E;
  if (S == E) {
    System.out.println("WARNUNG! Grade = Punkt!");
    System.out.println(S.toString());
    Lho.taste();
  }
  senkrecht = false;
  Steigung = 0.0;
  x1 = Startpunkt.getT();
  x2 = Endpunkt.getT();
  y1 = Startpunkt.getD();
  y2 = Endpunkt.getD();
  if (Lho.gleich(x1, x2)) {
    senkrecht = true;
  }
  else {
    Steigung = (y2 - y1) / (x2 - x1);
  }
  /* if (Steigung < 0 - Lho.eps) {
    System.out.println("WARNUNG! Gerade
```

```
     hat negative Steigung");
    System.out.println(toString());
    Lho.taste();
  } */
  used = true;
}

public boolean inUse() {
  return used;
}

//die Gerade wird durch zwei Geraden
gS und gZ ersetzt.
//-> die Gerade muss eine Konkatenation
von gS und gZ sein.
//dadurch wird die aktuelle Gerade
'uninteressant', wird aber
//nicht ganz gelöscht, damit sie nicht
später neu erzeugt wird.
public void drop(Gerade gS, Gerade gZ) {

  getStartpunkt().getSGeraden().remove(this);
  getStartpunkt().addSGerade(gS);
  getEndpunkt().getZGeraden().remove(this);
  getEndpunkt().addZGerade(gZ);
  used = false;

}

public StPkt getStartpunkt() {
  return Startpunkt;
}

public StPkt getEndpunkt() {
  return Endpunkt;
}

public boolean isSenkrecht() {
  return senkrecht;
}

public double getSteigung() {
  return Steigung;
}
```

```
public double getX1() {
  return x1;
}

public double getX2() {
  return x2;
}

public double getY1() {
  return y1;
}

public double getY2() {
  return y2;
}

//liefert den b-Wert der Geraden
in der Geradendarstellung
//g(x) = b+mx
//Gerade senkrecht -> 0.0 zurück (evtl.
 ungeschickt!!)
public double getB() {
  if (!senkrecht) {
    return (y1 - x1 * Steigung);
  }
  else {
    return 0.0;
  }
}

//liefert die Länge der Geraden
public double getLaenge() {
  return Math.sqrt( (x2 - x1) *
  (x2 - x1) + (y2 - y1) * (y2 - y1));
}

//Liefert den Funktionswert der Geraden
an der Stelle t
//Gerade an der Stelle nicht definiert
-> 0.0 zurück (!s.o.!)
//Gerade senkrecht -> maximaler Wert an
der Stelle t
public double get_d_at(double t) {
```

```
    if ( (t + Lho.eps < x1) || (x2 < t - Lho.eps)) {
      return 0.0;
    }
    else if (senkrecht) {
      return y2;
    }
    else {
      return y1 + Steigung * (t - x1);
    }
  }

  //zur Pfadverfolgung im Graphen
  public StPktList getPfad() {
    return Endpunkt.getPfad(this);
  }

  public StPktList getMinPfad() {
    return Endpunkt.getMinPfad(this);
  }

  public String toString() {
    return "g: " + Startpunkt.toString() +
     " ---> " + Endpunkt.toString()
        + " /S: " + Steigung + " " + senkrecht;
  }

  //Löscht die Gerade aus allen Listen,
   in denen sie vorkommt.
  //Somit hört sie auf zu existieren.
  public void suicide(GeradenList G) {
    getStartpunkt().getSGeraden().remove(this);
    getEndpunkt().getZGeraden().remove(this);
    G.remove(this);
  }
}
```

GeradenList.java

```
package markierung;

import java.util.Vector;
```

```
/*
 * GeradenList ist ein erweiterter Vector,
 der zum vereinfachten
 * Verwalten von Geraden dient und hier
 benötigte Funktionalitäten
 * zur Verfügung stellt.
 */

class GeradenList
    extends Vector {

  /*
   * fügt die Gerade G sortiert ein.
   * Sortierung: Aufwärts nach erstem Stützpunkt
   * bei Gleichheit: nach zweitem Stützpunkt.
   * existiert eine Gerade mit den gleiche
   Koordinaten bereits
   * in der Liste, wird diese zurückgegeben
   (und G nicht eingefügt.)
   * Ansonsten wird G selbst zurückgegeben.
   */
  public Gerade insert(Gerade G) {

    boolean iadd = true;
    boolean added = false;

    double x1_neu = G.getStartpunkt().getT();
    double x2_neu = G.getEndpunkt().getT();
    double y1_neu = G.getStartpunkt().getD();
    double y2_neu = G.getEndpunkt().getD();

    double x1_alt = 0;
    double x2_alt = 0;
    double y1_alt = 0;
    double y2_alt = 0;

    Gerade aG = G;

    int i;
    for (i = 0; i < size(); i++) {

      aG = (Gerade) get(i);
```

```
      x1_alt = aG.getStartpunkt().getT();
      x2_alt = aG.getEndpunkt().getT();
      y1_alt = aG.getStartpunkt().getD();
      y2_alt = aG.getEndpunkt().getD();

      if (x1_alt < x1_neu - Lho.eps) {
        continue;
      }
      if (Lho.gleich(x1_alt, x1_neu)) {
        if (y1_alt < y1_neu - Lho.eps) {
          continue;
        }
      }
      if ( (Lho.gleich(x1_alt, x1_neu)) &&
      (Lho.gleich(y1_alt, y1_neu))) {
        if (x2_alt < x2_neu - Lho.eps) {
          continue;
        }
      }
      if ( (Lho.gleich(x1_alt, x1_neu)) && (
      Lho.gleich(y1_alt, y1_neu))
          && (Lho.gleich(x2_alt, x2_neu))) {
        if (y2_alt < y2_neu - Lho.eps) {
          continue;
        }
      }

      added = true;

      if ( (G.getStartpunkt() != aG.getStartpunkt())
          || (G.getEndpunkt() != aG.getEndpunkt())) {
        add(i, G);
      }
      else {
        iadd = false;

      }
      break;
    }

    if (!added) {
      add(G);
    }
    if (!iadd) {
```

```
    return aG;
  }
  else {
    return G;
  }
}

//liefert die kürzeste der Geraden mit
 maximaler Steigung
public Gerade getBeste() {

  Gerade gmax = getGerade(0);
  double stmax = gmax.getSteigung();
  boolean gmaxSK = gmax.isSenkrecht();

  for (int i = 1; i < size(); i++) {
    Gerade g = getGerade(i);
    double st = g.getSteigung();
    boolean sk = g.isSenkrecht();
    if ( (!gmaxSK) && (sk || (st > stmax))) {
      gmax = g;
      stmax = st;
      gmaxSK = sk;
    }
    else if ( (Lho.gleich(st, stmax)) &&
    (!gmaxSK || sk)) {
      if (g.getLaenge() < gmax.getLaenge()) {
        gmax = g;
        stmax = st;
        gmaxSK = sk;
      }
    }
  }

  return gmax;
}

//liefert die kürzeste der Geraden mit
 minimaler Steigung
public Gerade getWorst() {

  Gerade gmin = getGerade(0);
  double stmin = gmin.getSteigung();
  boolean gminSK = gmin.isSenkrecht();
```

```
    for (int i = 1; i < size(); i++) {
      Gerade g = getGerade(i);
      double st = g.getSteigung();
      boolean sk = g.isSenkrecht();
      if (!sk && gminSK) {
          gmin = g;
          stmin = st;
          gminSK = sk;
      }
      else if (Lho.gleich(st, stmin) &&
      (!sk || gminSK)) {
        if (g.getLaenge() < gmin.getLaenge()) {
          gmin = g;
          stmin = st;
          gminSK = sk;
        }
      }
      else if (gminSK || ( (!sk) && (st < stmin))) {
        gmin = g;
        stmin = st;
        gminSK = sk;
      }
    }
    return gmin;
  }

  //liefert die i-te Gerade
  public Gerade getGerade(int i) {
    return (Gerade) get(i);

  }

}
```

Konjunktion.java

```
package markierung;

/*
 * Jede Konjunktion muss die Methoden calc
```

```
  und calc_neg erfüllen.
 * calc soll die korrekte Berechnung der
 T-Norm von A und B liefern.
 * calc_neg soll auch negative Werte zulassen,
  so dass das "Abschneiden"
 * an der 0 erst später erfolgen kann
 */
public abstract class Konjunktion {
  abstract double calc(double A, double B);

  abstract double calc_neg(double A, double B);
}
```

LabelKombinator.java

```
package markierung;

/*
     * Der Label-Kombinator ist die
     Kontroll-Klasse. die für alle Labelkombinationen
 * verantwortlich ist.
 */
public class LabelKombinator {

  private Konjunktion T;

  private LGraph derGraph;

  //Erzeugung des LK durch Übergeben der
   Konjuktion, mit der kombiniert werden
  //soll.
  public LabelKombinator(Konjunktion T) {
    this.T = T;
  }

  //Kombiniert die übergeben Label mittels
   T und gibt das Ergebnis zurück.
  public FLabel getIt(FLabel Foriginal,
  FLabel Goriginal) {

    //F und G sind Hilfslabel. Sind
    identisch zu ihren Ursprungslabeln,
```

```
    //allerdings werden in ihnen weitere
     Stützpunkte eingefügt.
    FLabel F = Foriginal.copy();
    FLabel G = Goriginal.copy();

    derGraph = new LGraph();

    //ts: Vector von möglicherweise
     interessanten t-Stellen.
    DVector ts = F.getTwerte();
    ts.addSome(G.getTwerte());

    //diese t-Stellen werden also zu
    F und zu G hinzugefügt.
    addSomeSP(F, ts);
    addSomeSP(G, ts);

    StPkt FSP1, FSP2, GSP1, GSP2;

    GeradenList glTemp1, glTemp2;

    glTemp1 = alleGeraden(F);
    glTemp2 = alleGeraden(G);

    /*
     * im Folgenden werden aus den
      Labelstützpunkten alle erlaubten
     * Geraden erzeugt und zum Graphen
     hinzugefügt. Der obere Hülle
     * des Graphen ist dann das Ziellabel
     */

    for (int i = 0; i < glTemp1.size(); i++) {
      Gerade g1 = glTemp1.getGerade(i);
      for (int j = 0; j < glTemp2.size(); j++) {
        Gerade g2 = glTemp2.getGerade(j);

if (Lho.verbose)
System.out.print("\r"+(i+1)+"/"+glTemp1.size()+"
 - "+(j+1)+"/"+glTemp2.size());

        FSP1 = g1.getStartpunkt();
        FSP2 = g1.getEndpunkt();
        GSP1 = g2.getStartpunkt();
```

```
        GSP2 = g2.getEndpunkt();
        machNeGerade(FSP1, GSP1, FSP1, GSP2);
        machNeGerade(FSP2, GSP1, FSP2, GSP2);
        machNeGerade(FSP1, GSP1, FSP2, GSP1);
        machNeGerade(FSP1, GSP2, FSP2, GSP2);
        machNeGerade(FSP1, GSP1, FSP2, GSP2);

      }
    }

    //abschneiden des Graphen an der
     y-Achse durch temporäres Hinzufügen
    //einer senkrechten Geraden an dieser Stelle.
    StPkt s01 = new StPkt(0.0, 1.0);
    derGraph.mslcheck=false;
    derGraph.insert(new StPkt(0.0, 0.0), s01);
    derGraph.kill(s01);
    derGraph.mslcheck=true;

    //nun ist der Graph vollständig und
    muss noch durchlaufen werden.
    //Die dabei besuchten Stützpunkte
    bilden das neue Label.
    StPktList daList = derGraph.getPfad();

    //"reinigen" des Graphen
    derGraph.clear();

    return new FLabel(daList);
  }

  //erstellt eine Liste mit allen möglichen Geraden
  private GeradenList alleGeraden(FLabel L) {
    Gerade Grtemp1, Grtemp2;
    StPkt Start, End;
    GeradenList GLT = new GeradenList();
    for (int i = 0; i < L.size() - 1; i++) {
      Start = L.getPktAt(i);
      End = L.getPktAt(i + 1);
      Grtemp1 = new Gerade(Start, End);
      GLT.insert(Grtemp1);

      int k = 2;
```

```
      if (i + k < L.size()) {
        End = L.getPktAt(i + k);
      }
      Grtemp2 = new Gerade(Start, End);

      while (i + k < L.size()
             && Lho.gleich(Grtemp1.getSteigung(),
              Grtemp2.getSteigung())) {

        GLT.insert(Grtemp2);
        k++;
        if (i + k < L.size()) {
          End = L.getPktAt(i + k);
        }
        Grtemp2 = new Gerade(Start, End);
      }
    }

    return GLT;

  }

  //fügt neue Stützpunkte in ein
  temporäres Label ein.
  private void addSomeSP(FLabel L, DVector D) {
    for (int i = 0; i < D.size(); i++) {
      double x = D.getd(i);
      double y = L.get_d_at(x);
      L.add(new StPkt(x, y));
    }
  }

  //kombiniert die übergebenen StPkte
  anhand der T-Norm.
  //Das Ergebnis wird in den Graphen
  eingefügt
  private void machNeGerade(StPkt A1,
  StPkt B1, StPkt A2, StPkt B2) {

    StPkt LSP1, LSP2;

    double x1 = T.calc_neg(A1.getT(), B1.getT());
    double y1 = Math.min(A1.getD(), B1.getD());
```

```
    LSP1 = new StPkt(x1, y1);

    double x2 = T.calc_neg(A2.getT(), B2.getT());
    double y2 = Math.min(A2.getD(), B2.getD());

    LSP2 = new StPkt(x2, y2);

    if (x2>=0-Lho.eps)
     derGraph.insert(LSP1, LSP2);
  }

  //liefert das MinLabel aus A und B.
  public FLabel getMin(FLabel A, FLabel B) {

   //System.out.println("min "+A+" / "+B);

    LGraph einGraph = new LGraph();
    //alle Geraden der Label werden in
     einen Graphen eingefügt und dabei
    //sofort auf Schnittpunkte überprüft.
    for (int i = 0; i < A.size() - 1; i++) {
      einGraph.insert(A.getPktAt(i),
      A.getPktAt(i + 1));
    }

    for (int i = 0; i < B.size() - 1; i++) {
      einGraph.insert(B.getPktAt(i),
      B.getPktAt(i + 1));
    }
    //das MinLabel besteht dann aus dem
    untersten Pfad dieses Graphen.
    FLabel C = new FLabel(einGraph.getMinPfad());
    //System.out.println("-> "+C);
    return C;
  }
}
```

LGraph.java

```
package markierung;
```

```
/*
 * LGraph ist ein Graph, aus dem das
  Label berechnet werden soll.
 * Der Graph entsteht durch ständiges
 Hinzufügen von Geraden.
 * Das Ziellabel wird durch eine Art
  DFS erzeugt.
 */
class LGraph {

  private int recTiefe;

  private double msl=0.1;
  public boolean mslcheck=true;
  private GeradenList GList;
  private StPktList SList;
  private StPkt nullpunkt;

  /*
   * Konstruktor:
   * es werd je eine globale Geraden-
   und Stützpunktliste erzeugt, in
   * denen jeder Punkt und jede Gerade

    genau einmal erfasst wird.
   * Der Nullpunkt muss in jedem Fall
   vorhanden sein und wird sofort
   * erzeugt. Hier beginnt später der
   "DFS".
   */
  public LGraph() {
    GList = new GeradenList();
    SList = new StPktList();
    nullpunkt = new StPkt(0.0, 0.0);
    SList.insert(nullpunkt);
  }

  //fügt SP in den Graphen ein Existiert
  ein Punkt mit den Koordinaten
  //bereits, wird dieser zurückgegeben und
  nichts eingefügt.
  public StPkt addSP(StPkt SP) {
    return SList.insert(SP);
  }
```

```
//liefert die Anzahl der Geraden im Graphen.
public int anzGeraden() {
  return GList.size();
}

//liefert die Anzahl der Stützpunkte im Graphen.
public int anzSP() {
  return SList.size();
}

/*
 * fügt die Punkte A und B (falls nicht
 bereits vorhanden) und die Gerade,
 * die diese Punkte verbindet, in den
 Graphen ein.
 * Gibt false zurück, falls diese Gerade
 bereits existiert oder
 * die Punkte identisch sind, sonst true.
 */
public boolean insert(StPkt A, StPkt B) {
  StPkt SP1 = SList.insert(A);
  StPkt SP2 = SList.insert(B);
  if (SP1 != SP2) {
    Gerade G = new Gerade(SP1, SP2);
    //wenn eine entsprechende Gerade
    bereits im Graphen existiert,
    //wird diese weiter betrachtet:
    Gerade g1 = GList.insert(G);
    //sollte g1 nicht bereits überflüssig
    sein, wird sie an
    //den Punkten in die entsprechenden
    Listen eingefügt.
    if (g1.inUse()) {
      SP1.addSGerade(g1);
      SP2.addZGerade(g1);
    }
    recTiefe = 0;
    //G==g1 falls die Gerade zum
    ersten Mal erzeugt wurde. In diesem
    //Fall muss sie auf Schnitte
    überprüft werden.
    if (G == g1) {
      schnitt(g1, 0);
```

```
      return true;
    }
    else {
      return false;
    }
  }
  else {
    return false;
  }
}

//diese Methode dient zum Einfügen
 von Geraden, die nicht auf
//Schnitte überprüft werden sollen.
Ansonsten siehe oben.
private boolean insert(Gerade G) {
  StPkt SP1 = G.getStartpunkt();
  StPkt SP2 = G.getEndpunkt();
  if (SP1 != SP2) {
    Gerade g1 = GList.insert(G);
    if (g1.inUse()) {
      SP1.addSGerade(g1);
      SP2.addZGerade(g1);
    }
    if (g1 == G && g1.inUse()) {
      return true;
    }
    else {
      return false;
    }
  }
  else {
    return false;
  }
}

//Entfernen des Punktes A und
aller Geraden, die ihn kannten.
public void kill(StPkt A) {
  A.suicide(GList);
  SList.remove(A);
}

//liefert den obersten Pfad von (0|0)
```

```
 bis (1|1).
public StPktList getPfad() {
  return nullpunkt.getPfad();
}

//liefert den untersten Pfad von
 (0|0) bis (1|1).
public StPktList getMinPfad() {
  return nullpunkt.getMinPfad();
}

//toString-Methoden eizelner
Teile des Graphen:
public String punkte() {
  return SList.toString();
}

public String geraden() {
  return GList.toString();
}

public String toString() {
  String S = "StPkte: " + punkte();
  S += "\nGeraden: " + geraden();
  for (int i = 0; i < SList.size();
  i++) {
    S += "\n Geraden im Punkt " +
    SList.getPkt(i).toString();
    S += "\n" + SList.getPkt(i).
    getZGeraden().toString();
    S += "\n" + SList.getPkt(i).
    getSGeraden().toString();
  }
  return S;
}

//zerstören aller Geraden,
damit SP-Listen leer sind.
public void clear() {
  while (!GList.isEmpty()) {
    GList.getGerade(GList.size() - 1)
    .suicide(GList);
  }
}
```

```
/*
 * Testet die Gerade Gneu auf Schnittpunkte
 mit den übrigen
 * Geraden des Graphen. start zeigt
 den Index an, ab dem die
 * Geradenliste durchsucht werden soll.
 * Wird ein Schnittpunkt gefunden,
 wird er zum Graphen hinzugefügt
 * und die Geraden müssen an dieser
 Stelle geteilt werden,
 */
public void schnitt(Gerade Gneu, int start) {

  Gerade w1 = null;
  Gerade w2 = null;

  StPkt S = null;
  StPkt a1, a2, n1, n2;

  int i;

  for (i = start; i < GList.size(); i++) {

    Gerade Galt = GList.getGerade(i);

    //teste nur mit Geraden, die noch
    interessant sind
    if (!Galt.inUse()) {
      continue;
    }

    StPkt St = schnitt(Gneu, Galt);

    if (St != null) {

      S = SList.insert(St);

      a1 = Galt.getStartpunkt();
      a2 = Galt.getEndpunkt();
      n1 = Gneu.getStartpunkt();
      n2 = Gneu.getEndpunkt();

      Gerade gtemp1, gtemp2;
```

```
//falls die alte Gerade
durch S geteilt wird....
if (a1 != S && a2 != S) {

  //füge die neuen Geraden hinzu...
  gtemp1 = new Gerade(a1, S);
  insert(gtemp1);
  gtemp2 = new Gerade(S, a2);
  insert(gtemp2);
  //und markiere die alte als
  uninteressant.
  //->gleichzeitig: ersetzung

  der alten Geraden durch
  //die neuen an den Stützpunkten.
  Galt.drop(gtemp1, gtemp2);
}

//dito für die neue Gerade:


//die hieraus neu entstehenden
 Geraden müssen aber

//erneut auf weitere Schnitte geprüft werden (s.u.).
if (n1 != S && n2 != S) {

  gtemp1 = new Gerade(n1, S);
  if (insert(gtemp1)) {
    w1 = gtemp1;
  }
  gtemp2 = new Gerade(S, n2);
  if (insert(gtemp2)) {
    w2 = gtemp2;

  }
  Gneu.drop(gtemp1, gtemp2);
}
else {
  continue; //? hat bisher gefehlt,
   aber nötig,
}
//oder ?!?! keine Fehler erzeugt ?!?!
```

```
        break; //Abbruch falls ein Schnitt
        gefunden
        // -> weiter mit den Teilgeraden.
      }
    }

    //wurde die neue Gerade geteilt,
    müssen die Teilgeraden weiter
    //untersucht werden.
    if (w1 != null) {
      schnitt(w1, i + 1); //Geraden bis
      Index i wurden bereits getestet
      recTiefe--;
    }
    if (w2 != null) {
      schnitt(w2, i + 1);
      recTiefe--;
    }

  }

  //ermittelt den Schnittpunkt der
  übergebenen Geraden.
  public StPkt schnitt(Gerade G1, Gerade G2) {
    if (mslcheck && (G1.getLaenge()<msl ||
    G2.getLaenge()<msl)) return null;
    StPkt S = null;

    if ( (G1.getStartpunkt() != G2.getStartpunkt())
        && (G1.getStartpunkt() != G2.getEndpunkt())
        && (G1.getEndpunkt() != G2.getStartpunkt())
        && (G1.getEndpunkt() != G2.getEndpunkt())) {

      if (G1.isSenkrecht() ^ G2.isSenkrecht()) {
        Gerade sk = G1;
        Gerade nsk = G2;
        if (G2.isSenkrecht()) {
          sk = G2;
          nsk = G1;
        }
        double x = sk.getStartpunkt().getT();
        double y = nsk.get_d_at(x);
        if (x >= nsk.getStartpunkt().getT()
```

```
            && x <= nsk.getEndpunkt().getT()) {

          if ( (y >= sk.getStartpunkt().getD())
              && (y <= sk.getEndpunkt().getD())) {
            S = new StPkt(x, y);

          }
        }
      }

      else if (G1.getSteigung() != G2.getSteigung()) {
        double xschnitt = (G2.getB() - G1.getB())
            / (G1.getSteigung() - G2.getSteigung());
if (xschnitt<=0-Lho.eps) return null;
        if ( (G1.get_d_at(xschnitt) != 0.0)
            && (G2.get_d_at(xschnitt) != 0.0)) {
          S = new StPkt(xschnitt, G1.get_d_at(xschnitt));
        }
      }
    }

    return S;
  }

}
```

Lho.java

```
package markierung;

import java.io.*;
import java.text.*;
import markierung.*;

/*
 * Lho stellt einige Hilfsoperationen zur
  verfügung. Alle Methoden
 * und Attribute sind static, Lho soll
 nicht instantiiert werden,
 */
```

```
public class Lho {

  //true, falls an einigen Stellen auf einen
   Tastendruck gewartet werden
  //soll (bei Benutzung von GUI sollte immer
  false eingestellt sein).
  public static boolean tastendruck = false;
  public static boolean verbose = false;

  //eps soll rundungsfehler vermeiden (s.u.)
  public static double eps = 0.0001;

  //liefert eine ordentliche gerundete
  Stringdarstellung von d.
  public static String dez(double d) {
    DecimalFormat df = new DecimalFormat("#.###");
    return df.format(d);
  }

  //wartet auf Tastendruck, falls tastendruck
   (s.o.) entsprechend gesetzt.
  public static void taste() {
    if (tastendruck) {
      try {
        System.in.read();
        System.in.read();
      }
      catch (Exception e) {
        System.out.println(e.toString());
      }
    }
  }

  public static void taste(boolean b) {
    if (b) {
      try {
        System.in.read();
        System.in.read();
      }
      catch (Exception e) {
        System.out.println(e.toString());
      }
    }
  }
```

```
//überprüft a und b auf Gleichheit.
//a gleich b falls ihr Abstand < eps ist.
public static boolean gleich(double a, double b) {
  if (Math.abs(a - b) < eps) {
    return true;
  }
  else {
    return false;
  }
}

//Hilfsklasse zur Eingabe von Integer-werten.
public static int inputi(String text) {
  int a = 0;
  try {
    BufferedReader din
        = new BufferedReader(new
        InputStreamReader(System.in));
    System.out.print(text);
    a = Integer.parseInt(din.readLine());
  }
  catch (Exception e) {
    System.out.println(e.toString());
  }
  return a;
}

//Hilfsklasse zur Eingabe von Double-werten.
public static double inputd(String text) {
  double a = 0;
  try {
    BufferedReader din
        = new BufferedReader(new
        InputStreamReader(System.in));
    System.out.print(text);
    a = Double.parseDouble(din.readLine());
  }
  catch (Exception e) {
    System.out.println(e.toString());
  }
  return a;
}
```

```
}
```

StPkt.java

```
package markierung;

public class StPkt {

  private double t, d;
  private GeradenList GLStart;
  private GeradenList GLZiel;

  public StPkt(double t, double d) {

    this.t = t;
    this.d = d;
    if (d < 0) {
      System.out.println("!!!!!! d<0 !!!!!!!!");
      System.out.println(toString());
      Lho.taste();
    }
    GLStart = new GeradenList();
    GLZiel = new GeradenList();
  }

  //liefert t-Koordinate des Punktes
  public double getT() {
    return t;
  }

  //liefert d-Koordinate des Punktes
  public double getD() {
    return d;
  }

  public String toString() {
    return "(" + Lho.dez(t) + "|" +
    Lho.dez(d) + ")";
  }

  public String getString() {
```

```
   return Lho.dez(t)+" "+Lho.dez(d);
  }

  public void addSGerade(Gerade G) {
    GLStart.insert(G);
  }

  public void addZGerade(Gerade G) {
    GLZiel.insert(G);
  }

  public GeradenList getSGeraden() {
    return GLStart;
  }

  public GeradenList getZGeraden() {
    return GLZiel;
  }

  public StPktList getPfad(Gerade letzte) {
/*
    if (Lho.debug_Pfad) {
      System.out.println(toString());
      System.out.println(GLStart.toString());
      if (!GLStart.isEmpty()) {
        System.out.println(GLStart.getBeste().

        toString());
      }
      Lho.taste();
    }

*/
    StPktList s;
    if (GLStart.isEmpty()) {
      s = new StPktList();
      s.add(0, this);
    }
    else {
      Gerade ng = GLStart.getBeste();
      s = ng.getPfad();
      if (! (Lho.gleich(ng.getSteigung(),
      letzte.getSteigung()))
          || (ng.isSenkrecht() != letzte.
```

```
          isSenkrecht())) {
        s.add(0, this);
      }
    }
    return s;
  }

  public StPktList getMinPfad(Gerade letzte) {
/*
    if (Lho.debug_Pfad) {
      System.out.println(toString());
      System.out.println(GLStart.toString());
      if (!GLStart.isEmpty()) {
        System.out.println(GLStart.getWorst()
        .toString());
      }
      Lho.taste();
    }
*/
    StPktList s;
    if (GLStart.isEmpty()) {
      s = new StPktList();
      s.add(0, this);
    }
    else {
      Gerade ng = GLStart.getWorst();
      s = ng.getMinPfad();
      if (! (Lho.gleich(ng.getSteigung(),
      letzte.getSteigung()))
          || (ng.isSenkrecht() != letzte.
          isSenkrecht())) {
        s.add(0, this);

      }
    }
    return s;
  }

  public StPktList getPfad() {
/*
    if (Lho.debug_Pfad) {
      System.out.println(toString());
      System.out.println(GLStart.toString());
      if (!GLStart.isEmpty()) {
```

```
        System.out.println(GLStart.getBeste()
        .toString());
      }
      Lho.taste();
    }
*/
    StPktList s = GLStart.getBeste().getPfad();
    s.add(0, this);
    return s;
  }

  public StPktList getMinPfad() {
/*
    if (Lho.debug_Pfad) {
      System.out.println(toString());
      System.out.println(GLStart.toString());
      if (!GLStart.isEmpty()) {
        System.out.println(GLStart.getWorst()
        .toString());
      }
      Lho.taste();
    }
*/
    StPktList s = GLStart.getWorst().getMinPfad();
    s.add(0, this);
    return s;
  }

  public void suicide(GeradenList GL) {
    while (GLStart.size() != 0) {
      GLStart.getGerade(0).suicide(GL);
    }
    while (GLZiel.size() != 0) {
      GLZiel.getGerade(0).suicide(GL);
    }
  }

  public void setLocation(StPkt s) {
    this.t = s.t;
    this.d = s.d;
  }

  public void setLocation(double t,
   double d) {
```

```
    this.t = t;
    this.d = d;
  }

}
```

StPktList.java

```
package markierung;

import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

public class StPktList
    extends Vector
    implements ListModel {

  private LinkedList listeners;
  private ListDataEvent lde;

  public StPktList() {
    listeners = new LinkedList();
    lde = new ListDataEvent(this,
                            javax.


       swing.
         event.ListDataEvent.CONTENTS_CHANGED, 0,
                            0);
  }

  public StPkt insert(StPkt S) {

    boolean addplease = (S != null);
    int i = 0;
    double t_alt, t_neu, d_alt, d_neu;

    if (addplease) {
```

```
/*
      if (Lho.debug_insertSP) {
        System.out.println("Liste vorher: "
         + toString());
        System.out.println("insert Pkt  : "
        + S.toString());
      }
*/
      t_neu = S.getT();
      d_neu = S.getD();

      for (i = 0; i < size(); i++) {

        t_alt = getPkt(i).getT();
        d_alt = getPkt(i).getD();

        if (Lho.gleich(t_alt, t_neu)) {
          if (Lho.gleich(d_alt, d_neu)) {
            return getPkt(i);
          }
          if (d_alt > d_neu) {
            break;
          }
        }
        else if (t_alt > t_neu) {
          break;
        }

      }

    }

    if (addplease) {
      add(i, S);

      fireModelChanged();

    }
/*
    if (Lho.debug_insertSP) {
      System.out.println("Liste nachher:
      " + toString());
      Lho.taste();
    }
```

```
*/
    return S;

  }

  public void insertChecked(StPkt s) {
    if ((s.getT() < 0.0) || (s.getT() > 1.0)
    || (s.getD() < 0.0) || (s.getD() > 1.0))
      return;
    else
      insert(s);
  }

  public Object remove(int i) {
    if ( (i > 0) && (i < this.size() - 1)) {
      Object o = super.remove(i);
      this.fireModelChanged();
      return o;
    }
    else {
      return null;
    }
  }

  public void add(int i, Object o) {
    super.add(i, o);
    this.fireModelChanged();
  }

  public StPkt getPkt(int i) {
    return (StPkt) get(i);
  }

  public void addListDataListener
  (ListDataListener l) {
    listeners.add(l);
  }

  public Object getElementAt(int index) {
    return get(index);
  }

  public int getSize() {
    return this.size();
```

```
  }

  public void removeListDataListener
  (ListDataListener l) {
    listeners.remove(l);
  }

  public void fireModelChanged() {
    for (int i = 0; i < listeners.size(); i++) {
      ( (ListDataListener) listeners.get(i))
      .contentsChanged(lde);
    }
  }

  public Object clone() {
    StPktList newStPktList = new StPktList();
    Iterator i = this.iterator();
    StPkt st;

    while (i.hasNext()) {
      st = (StPkt)i.next();
      newStPktList.add(new StPkt(st.getT(),
      st.getD()));
    }

    return newStPktList;
  }

}
```

Und_bold.java

```
package markierung;

//die Bold-Konjuktion.....
public class Und_bold
    extends Konjunktion {

  public double calc(double A, double B) {
    return Math.max(0.0, A + B - 1.0);
  }
```

```
  public double calc_neg(double A, double B) {
    return A + B - 1.0;
  }
}
```

Und_mal.java

```
package markierung;

/* Bei Benutzung von und_mal liefert der
 LabelKombinator keine korrekten
 * Ergebnisse, da Zielfkt. nicht mehr
  stückweise linear ist. */
public class Und_mal
    extends Konjunktion {

  public double calc(double A, double B) {
    return A * B;
  }

  public double calc_neg(double A, double B) {
    return A * B;
  }
}
```

Und_min.java

```
package markierung;

//die Min-Konjuktion.....
public class Und_min
    extends Konjunktion {

  public double calc(double A, double B) {
    return Math.min(A, B);
  }

  public double calc_neg(double A, double B) {
    return Math.min(A, B);
  }
```

```
}
```

9.3 Sourcen des Parsers

```
package parser;

import java.util.Vector;

public class Parser {

  /*--------------------------
  Version 1.0,   28.10.03, ms, jdk: 1.3.1_02
  Version 1.1,   30.10.03, ms: Integration der Formelklassen
  Version 1.2,   03.11.03, ms: PL wird jetzt geparst, einige
  Einschränkungen
  Version 1.3,   19.11.03, ms: PL wird jetzt komplett geparst
  Version 1.3.1, 01.12.03, ms: kleinere Verbesserungen,
  Debug-Modus
  Version 1.3.2, 03.12.03, ms: Integration des Laders
  Version 1.3.3, 10.12.03, ms: gleichnamige Konstanten
  und gleichnamige ungebundene Variablen
                               innerhalb einer Formel sind
                                jetzt gleiche;
  Version 1.4,   17.12.03, ms: gleichnamige Funktionen
  innerhalb einer Formel sind jetzt gleiche Objekte
                               Axiome werden geparst,
                                gleichnamige FVariablen
                                 sind gleiche Objekte,

                               gleichnamige nicht-freie
                               Variablen ebenfalls
  Version 1.4.1, 18.12.03, ms: Bug beim Setzen der Sorten
   von Variablen in Quantoren behoben
  Version 1.4.2, 26.01.04, ms: getNichtFrei-Methode
  vom Lader integriert
                               Beispiel: (@x)(A>B)>
                               (A>(@x)B) : x A //x
                               nicht frei in A

       Hierbei ist das "Quantor-x" dasselbe
```

```
     Objekt wie das nicht-freie-Variablen-Objekt,
     das in der FVariablen A referenziert wird !

Version 1.4.3, 17.02.04, ms: Bugfixes: o KBs dürfen
jetzt auch FVariablen enthalten
                      o Quantorvariablen
                         dürfen jetzt auch Länge > 1 haben

Version 1.4.4, 09.03.04, ms: o (hoffentlich)
überflüssigen Testcode entfernt
                            o Debug-Modus ausgeschaltet

Erzeugt Ausgabe für graphviz
( www.research.att.com/sw/tools/graphviz/ )
Aufruf unter Unix (bash) z.B. mit java Parser
| dot -Tps -o out.ps; gv out.ps
Achtung: Das Parsen der Prädikate wird in
diesem Formelbaum NICHT dargestellt.
--------------------------*/

private static Vector vec,vvec,cvec,fvec;
//Vector für Prädikate, Variablen,
 Konstanten und Funktionen
private static Vector fvvec;
//Vector für FVariablen
public static boolean d=false; //der
Debug-Modus ist standardmäßig ausgeschaltet
private static Lader lader=new Lader(); //notwendig,
da einige Methoden in Lader nicht "static" sind
private static boolean isAxiom=false;

public static Formel parseAxiom(String f) {
  fvvec=new Vector(10,5); //initial capacity
  and capacity increment
  vvec=new Vector(10,5); //zur Speicherung
  der nicht-freien Variablen sowie der
  Variablen in Quantoren
  isAxiom=true;         //XXX nicht Thread-safe!!
  Formel temp = parse(f);
  isAxiom=false;
  return temp;
}

public static Formel parse(String f) {
```

```
//ein Wrapper, um (1.) den Vector zu löschen und
                                          //um
                   (2.)
                     removeAllWhiteSpace
                     nicht mehrmals aufrufen zu müssen
  vec=new Vector(10,5); //initial capacity and
  capacity increment
  vvec=new Vector(10,5);
  cvec=new Vector(10,5);
  fvec=new Vector(10,5);
  fvvec=new Vector(10,5); //in einer
  gespeicherten KB können auch FVariablen enthalten sein
  f=removeAllWhiteSpace(f);
  if (d) System.out.println("digraph g {");
  Formel temp = parseRec(f);
  if (d) System.out.println("}");
  return temp;
}

public static Formel parseRec(String f) { //die
eigentliche rekursive Parse-Methode
  String[] arr={"", f};
  String[] arr2={"","",f};

  arr = parseOperator(f,">");
  if (!arr[1].equals(f)) {
    if (d) System.out.println("\""+f+"\" ->
    \""+trim(arr[0])+"\";");
    if (d) System.out.println("\""+f+"\" ->
    \""+trim(arr[1])+"\";");
    return new KonjFormel( parseRec(arr[0]),
    parseRec(arr[1]),">" );
  }

  arr = parseOperator(f,"|");
  if (!arr[1].equals(f)) {
    if (d) System.out.println("\""+f+"\" ->
     \""+trim(arr[0])+"\";");
    if (d) System.out.println("\""+f+"\" ->
     \""+trim(arr[1])+"\";");
    return new KonjFormel( parseRec(arr[0]),
    parseRec(arr[1]),"|" );
  }
```

```
arr = parseOperator(f,"&");
if (!arr[1].equals(f)) {
  if (d) System.out.println("\""+f+"\" ->
   \""+trim(arr[0])+"\";");
  if (d) System.out.println("\""+f+"\" ->
  \""+trim(arr[1])+"\";");
  return new KonjFormel( parseRec(arr[0]),
  parseRec(arr[1]),"&" );
}

String temp = trim(f);
if (!temp.equals(f)) {
  return parseRec(temp);
}

arr = parseNegation(f);
if (!arr[1].equals(f)) {
  if (d) System.out.println("\""+f+"\" ->
  \""+trim(arr[1])+"\";");
  return new NegFormel(parseRec(arr[1]),"!");
}

Sorte[] so=new Sorte[1]; //Variablen in
Quantoren sollen auch eine Sorte haben

arr2 = parseQuantor(f,"?");
if (!arr2[2].equals(f)) {
  if (!isAxiom) so = lader.getSorten(arr2[1]);
  else so[0]=new Sorte("-- keine Sorte --");
  if (d) System.out.println("\""+f+"\" ->
  \""+trim(arr2[2])+"\";");
  return new QuantorFormel("?",
  generateVariable(arr2[1], so[0]),
  parseRec(arr2[2])); //String quantor,Variable v,
  Formel f
}

arr2 = parseQuantor(f,"@");
if (!arr2[2].equals(f)) {
  if (!isAxiom) so = lader.getSorten(arr2[1]);
  else so[0]=new Sorte("-- keine Sorte --");
  if (d) System.out.println("\""+f+"\" ->
  \""+trim(arr2[2])+"\";");
  return new QuantorFormel("@", generateVariable(arr2[1],
```

```
      so[0]),parseRec(arr2[2])); //String quantor,Variable
      v, Formel f
    }

    //hier sind wie an einem Blatt!
    //hinter Aussagenvariablen mit gleichen Namen
    sollen die gleichen
    //Objekte "stecken":

    if (!isAxiom) { //wenn wir gerade KEIN Axiom parsen,
    dann...
      for (int i=0; i<vec.size(); i++) {
        Praedikat p = (Praedikat)vec.get(i);
        if (p.toString().equals(f)) {
          return p; //XXX in der PL nur sinnvoll für
          Prädikate ohne Variablen, d.h. nur mit
          Grundtermen
        }           //XXX und für Prädikate mit
        nicht durch Quantoren gebundene Variablen
      }

      //nicht drin, also neu erzeugen und in den
       Vector packen:

      //XXXX f kann auch eine FVariable sein

      Praedikat p = parsePraedikat(f);
      if (p==null) return generateFVariable(f);

      vec.add(p);
      return p;
    }
    else { //wir parsen ein Axiom: Anstelle von
    Prädikaten parsen wir FVariablen
     return generateFVariable(f); //z.B. f="A"
    }
  }

  private static Praedikat parsePraedikat(String f) {
    //z.B. f = "Rot(fi(x1,c2),y3)", alle
    Blanks schon entfernt
    //Prädikat abschneiden:
    String p = "<<not initialized>>";
    for (int i=0; i<f.length(); i++) {
```

```
  String temp = f.substring(i,i+1);
  if (temp.equals("(")) {
    p = f.substring(0,i); //das Praedikat
    f = f.substring(i+1,f.length()-1); //Klammern
    () gleich mit abschneiden
    break; //raus hier
  }
}

/* XXXX Wenn das Praedikat f in Wirklichkeit
 eine FVariable ist, z.B. "B", dann
 * ist p="<<not initialized>>" und f="B".
 */

if (p.equals("<<not initialized>>"))
  return null;

//jetzt f = fi(x1,c2),y3
//Kommata sind Trennzeichen, wenn sie nicht
innerhalb von Klammern stehen:
f += ","; //ein dirty trick
int auf=0, zu=0; //Klammern zählen
Term[] buffer=new Term[50]; //ein Buffer für
 die Terme
int count=0, next=0;
for (int i=0; i<f.length(); i++) {
  String temp = f.substring(i,i+1);
  if (temp.equals("(")) auf++;
  if (temp.equals(")")) zu++;
  if (temp.equals(",") && auf==zu) { //wir
  sind an einem Komma
    String name = f.substring(next,i);
    Term term = new Konstante("<<not initialized>>",
     new Sorte("<<not initialized>>"));
    Sorte[] sorten=lader.getSorten(name);
    if (lader.isKonstante(name)) term =
    generateKonstante(name, sorten[0]);
    else if (lader.isVariable(name)) term =
     generateVariable(name, sorten[0]);
    else term = parseFunktion(name); //
    dirty trick: also eine Funktion
    buffer[count++]=term;
    next=i+1;
  }
```

```
    }

    //jetzt buffer={"fi(x1,x2)","y3","", ...}
    //Buffer trimmen:
    Term[] terme = new Term[count];
    for (int i=0; i<count; i++) terme[i]=buffer[i];
    return new Praedikat(p, terme);
  }

   private static FVariable generateFVariable
   (String name) {
    FVariable found = null, term = null;
    for (int j=0; j<fvvec.size(); j++) {
      FVariable v = (FVariable)fvvec.get(j);
      if (v.toString().equals(name)) {
        found=v;
        break; //können jetzt raus aus der
        for-Schleife
      }
    }
    if (found==null) {
      String nichtFrei=Lader.getNichtFrei(name);
      //XXX Was passiert, wenn nichtFrei = null gesetzt wird ?

      Variable nichtFreiVar = null; //XXX Das
      passiert!
      if (nichtFrei!=null) nichtFreiVar =
      generateVariable(nichtFrei, null);

      term = new FVariable(name, nichtFreiVar);
      fvvec.add(term); //und rein damit in den Vektor
    }
    else term=found;
    return term;
  }

  private static Variable generateVariable(String
  name, Sorte sorte) {
    Variable found = null, term = null;
    for (int j=0; j<vvec.size(); j++) {
      Variable v = (Variable)vvec.get(j);
      if (v.toString().equals(name)) {
        found=v;
        break; //können jetzt raus aus der
```

```
      for-Schleife
    }
  }
  if (found==null) {
    term = new Variable (name, sorte);
    vvec.add(term); //und rein damit in den Vektor
  }
  else term=found;
  return term;
}

private static Konstante generateKonstante(String name,
 Sorte sorte) {
  Konstante found = null, term = null;
  for (int j=0; j<cvec.size(); j++) {
    Konstante v = (Konstante)cvec.get(j);
    if (v.toString().equals(name)) {
      found=v;
      break; //können jetzt raus aus der for-Schleife
    }
  }
  if (found==null) {
    term = new Konstante(name, sorte);
    cvec.add(term); //und rein damit in den Vektor
  }
  else term=found;
  return term;
}

private static Funktion generateFunktion(String
 funktion, String name, Term[] terme, Sorte sorte) {
  Funktion found = null, term = null;
  for (int j=0; j<fvec.size(); j++) {
    Funktion f = (Funktion)fvec.get(j);
    if (f.toString().equals(funktion)) {
      found=f;
      break; //können jetzt raus aus der
      for-Schleife
    }
  }
  if (found==null) {
    term = new Funktion(name, terme, sorte);
    fvec.add(term); //und rein damit in den Vektor
  }
```

```
    else term=found;
    return term;
  }


  private static Funktion parseFunktion(String f) {
    //z.B. f = "fi(fi(x1,c2),y3)", alle Blanks
    schon entfernt
    //Funktion abschneiden:
    String f_save = f;
    String p = "<<not initialized>>";
    for (int i=0; i<f.length(); i++) {
      String temp = f.substring(i,i+1);
      if (temp.equals("(")) {
        p = f.substring(0,i); //die Funktion
        f = f.substring(i+1,f.length()-1); //Klammern
        () gleich mit abschneiden
        break; //raus hier
      }
    }

    //jetzt f = fi(x1,c2),y3
    //Kommata sind Trennzeichen, wenn sie nicht
     innerhalb von Klammern stehen:
    f += ","; //ein dirty trick
    int auf=0, zu=0; //Klammern zählen
    Term[] buffer=new Term[50]; //ein Buffer
    für die Terme
    int count=0, next=0;
    for (int i=0; i<f.length(); i++) {
      String temp = f.substring(i,i+1);
      if (temp.equals("(")) auf++;
      if (temp.equals(")")) zu++;
      if (temp.equals(",") && auf==zu) {
      //wir sind an einem Komma
        String name = f.substring(next,i);
        Term term = new Konstante("<<not
        initialized>>", new Sorte("<<not initialized>>"));
        Sorte[] sorten=lader.getSorten(name);
        if (lader.isKonstante(name)) term =
         generateKonstante(name, sorten[0]);
        else if (lader.isVariable(name))  term =
        generateVariable (name, sorten[0]);
        else term = parseFunktion(name); //ein dirty trick
```

```
      buffer[count++]=term;
      next=i+1;
    }
  }

  //jetzt buffer={"fi(x1,x2)","y3","", ...}
  //Buffer trimmen:
  Term[] terme = new Term[count];
  for (int i=0; i<count; i++) terme[i]=buffer[i];
  return generateFunktion(f_save, p, terme,
   lader.getSorten(p)[0]);
}

private static String[] parseOperator(String f,
 String op) {
  int auf=0, zu=0; //Klammern zählen
  for (int i=0; i<f.length(); i++) {
    String temp = f.substring(i,i+1);
    if (temp.equals("(")) auf++;
    if (temp.equals(")")) zu++;
    if (temp.equals(op) && auf==zu) {
      String[] ret = {f.substring(0,i), f.substring(i+1)};
      return ret;
    }
  }
  String[] ret = {"",f};
  return ret;
}

private static String[] parseNegation(String f) {
  if (f.startsWith("!")) {
    String[] ret = {"!", f.substring(1)};
    return ret;
  }
  else {
    String[] ret = {"", f};
    return ret;
  }
}

private static String[] parseQuantor(String f,
String op) {
  //Formel ist schon getrimmt, muss also mit
  "(" anfangen
```

```
  if (f.startsWith("(")) {
    //vorher: String temp  = (f.substring(1)).
    trim(); //Klammer abschneiden und Rest trimmen
    String temp  = f.substring(1);
    if (temp.startsWith(op)) {
      //vorher: temp = (temp.substring(1)).trim();
      //Operator abschneiden und Rest trimmen
      temp = temp.substring(1);
      //XX Die Variable endet mit ")"
      //XX Bug: String var = temp.substring(0,1);
       //das erste Zeichen ist die Variable
      //XX besser:
      String var=null;
      for (int i=0; i<temp.length(); i++) {
        String str = temp.substring(i,i+1);
        if (str.equals(")")) {
          var=temp.substring(0,i);
          temp=temp.substring(i+1);
}
        if (var!=null) break; //bloß raus hier
      }
      //vorher: temp = (temp.substring(1)).trim();
       //Variable abschneiden und Rest trimmen
      //vorher: temp = temp.substring(1);
      String[] ret = {op, var, temp};
      return ret;
    }
  }
  //falls die Formel nicht mit "(" anfängt,
   keine Quantorformel
    String[] ret = {"","", f};
    return ret;
}

private static String removeAllWhiteSpace(String s) {
  for (int i=0; i<s.length(); i++) {
    String temp = s.substring(i,i+1);
    if (temp.equals(" "))
      return removeAllWhiteSpace(s.substring(0,i)
      +s.substring(i+1));
      //Stelle i ist leerzeichen; schnipp,schnapp:
      leerzeichen ab
      //aus dem neuen String alle Leerzeichen löschen
  }
```

```
    return s;
  }

  private static String trim(String f) { //überflüssige
  Klammern entfernen
    int auf=0, zu=0; //Klammern zählen
    for (int i=0; i<f.length(); i++) {
      String temp = f.substring(i,i+1);
      if (temp.equals("(")) auf++;
      if (temp.equals(")")) zu++;
      if (auf==zu && i<f.length()-1) return f;
    }
    if (f.startsWith("(") && f.endsWith(")") && auf==zu)
     return trim(f.substring(1,f.length()-1));
    else return f;
  }

//GUI-Hilfsmethode für Umwandlung der math. Zeichen in
 Unicode

  public static String sonderzeichenErsetzungsmethode
  (String f) {
    int andre=0;
    char alex;
    StringBuffer hanns = new StringBuffer(1025); //Hanns
    wollte 1001. Wir aber nicht!!

    for (andre=0; andre < f.length(); andre++){
      alex = f.charAt(andre);
      switch (alex){
        case '@': hanns.append('\u2200');
          break;
        case '?': hanns.append('\u2203');
          break;
        case '&': hanns.append('\u2227');
          break;
        case '|': hanns.append('\u2228');
          break;
        case '>': hanns.append('\u21D2');
          break;
       default : hanns.append(alex);
      }
    }
    return hanns.toString();
```

```
  }

}


package parser;

import java.util.Vector;
import markierung.*;
import verarbeitung.*;
import java.lang.*;
import java.lang.reflect.Array;

public class Existenzquantor
    extends Quantor {

    private boolean d = false;

    public Existenzquantor(){
;
    }

    public String toString(){
return "?";
    }

/**
     * Diese Methode wird bei der Überprüfung von
      Existenzquantoren verwendet.
 * Übergeben wird eine konjugierte Formel
 (Implikation), welche auf der linken Seite
 als Prämisse eine quantorisierte Formel enthält.
 * In dieser quantorisierten Formel werden für
  die vom Quantor abhängige Variable sukzessive
  alle Individuen der zugehörigen Sorte eingesetzt
 * und überprüft, ob diese Formel in der Wissensbasis
 vorhanden ist. Um den Existenzquantor zu erfüllen
  muss lediglich ein Individuum ein positives Resultat
    * liefern. Dies enspricht einer Disjunktion
     über alle Individuen eingesetzt in die
     quantorisierte Formel.
 * Die Bewertung der quantorisierten Formel ergibt
  sich aus dem FLabel des "besten" Summanden bei
  der Einsetzung. Diese Bewertung wird zurückgegeben
```

```
 * und dient als Bewertung für die Konklusion der
 übergebenen Implikation.
 * @param fcopy Implikation, welche auf der
 linken Seite eine quantorisierte Formel enthält.
 * @param kb Wissensbasis, welche auf jeden einzelnen
 Summanden überprüft wird.
 * @return gibt ein FLabel zurück, welches als
  Bewertung für die Konklusion der übergebenen
  Implikation dient.
 * @exception GefundenException nur Weiterleitung
 der Exception
    */

  public FLabel erfuellt(Formel fcopy, Vector kb)
  throws GefundenException{
      QuantorFormel qf=(QuantorFormel) ((KonjFormel)
      fcopy).getLeft();
      // Existenzquantor->Disjunktion über alle
      Individuen der Sorte
      boolean erfuellt = false;
      Konstante [] individuen = qf.getVariable().
      getSorte().getKonstanten();
      FLabel temp = new FLabel();
      //Beim Existenzquantor beginnt man mit
      unbekannt und ersetzt immer dann, wenn
      bessere Bewertungen gefunden wurden
      temp.set_Unknown();
if (d) System.out.println("ExQuantor: "+
qf.toString2());
      for (int i=0;i<individuen.length;i++)
      {
if (d) System.out.println("einsetzen von
"+individuen[i]);
  //Der Wirkungsbereich des Quantors wird
  für alle Individuen überprüft
  QuantorFormel temptesting = (QuantorFormel)
  InfTool.kopiere(qf); //kopieren
  Variable gesucht = temptesting.getVariable();
   //gebundene Variable bestimmen
         Formel testing = temptesting.getFormel();
  InfTool.varDurchKonstErsetzen(testing, gesucht,
   individuen[i]); //in Kopie einsetzen
  if (d) System.out.println("eingesetzt in "+testing);
         MarkFormel bewFormel = InfTool.suchInKB_q
```

```
          (testing, fcopy); //in KB suchen
  qf=(QuantorFormel) ((KonjFormel)fcopy).getLeft();
  if ((bewFormel == null) && (testing instanceof
  KonjFormel)) bewFormel = InfTool.suchInKBund(testing);
     //andere Möglichkeit zu suchen
          if (bewFormel == null) {
   if (d) System.out.println("nicht gefunden");
   continue;
}
          else{
   if (d) System.out.println("gefunden");
      temp = temp.min(bewFormel.getFLabel());
      //solange Formeln gefunden werden, wird
       sich die beste Bewertung gemerkt
      erfuellt = true; //es reicht
       eine gefundene Formel aus damit
       der Existenzquantor erfüllt ist.
          }

      }

      if(erfuellt)return temp;
      else return null;

  }

  public boolean gleicherQuantor(Quantor q) {
   if (q instanceof Existenzquantor) return true;
else return false;
  }

}
*****************************************

package parser;

import java.io.*;
import parser.*;

// eine Prädikatenlogische Formel !!!

public abstract class Formel
    implements Serializable {
```

```
abstract public String toString2();

abstract public int laenge();

}

*******************************************


package parser;

import java.util.Vector;

//ist im Prinzip ein Vector

public class Formelmenge
{
  Vector v;
  public Formelmenge(Vector v)
  {
   this.v = v;

  }

  public Vector getFormelmenge() {return v;};
  public void setFormelmenge(Vector v) {this.v = v;};

  public String toString()
  {
    String ret = "";
    for(int i = 0; i< v.size(); i++) ret = ret +
    v.get(i).toString() + "\n";
    return ret;
  }
}

*******************************************

package parser;

import java.util.Vector;

//ist im Prinzip ein Vector
```

```
public class Formelmenge
{
  Vector v;
  public Formelmenge(Vector v)
  {
   this.v = v;

  }

  public Vector getFormelmenge() {return v;};
  public void setFormelmenge(Vector v) {this.v =
  v;};

  public String toString()
  {
    String ret = "";
    for(int i = 0; i< v.size(); i++) ret = ret +
     v.get(i).toString() + "\n";
    return ret;
  }
}


************************************************

/* Generated by Together */

package parser;

//diese Klasse repräsentiert einen prädikatenlogische
Funktion

public class Funktion
    extends Term {
  private int arity;
  private Term[] terme;

  public Funktion(String name, Term[] terme, Sorte
  sorte) {
    if (terme == null) {
      System.out.println("Achtung: Fehler: funktion
      ohne Term-Parameter !");
    }
    arity = terme.length;
```

```
    super.name = name;
    this.terme = terme;
    super.sorte = sorte;
  }

  public Term getTermNr(int i) {
    i--;
    if ( (i < 0) || (i >= arity)) {
      return null;
    }
    else {
      return terme[i];
    }
  }

  public void setTermNr(int i, Term t) {
   i--;
terme[i]=t;
  }

  public int getArity() {
    return arity;
  }

  public String toString() {
    String ret;
    ret = name + "(";

    for (int i = 1; i < arity + 1; i++) {
      ret = ret + getTermNr(i).toString();
      if (i != arity) {
        ret = ret + ",";
      }
    }
    return ret + ")";
  }

  public String toString2() {
    return name;
  }
}
****************************************

package parser;
```

```
/* Generated by Together */

// eine sogenannte Formel-Variable !!!!!

public class FVariable
    extends Formel {

  String name="FVariable ohne Namen";
  boolean e;
  Formel durch;
  Variable nichtEnthalten;
  static int laengenwert=1;

  public FVariable() {
    e = false;
    durch = null;
nichtEnthalten=null;
  }

  public FVariable(Variable v) {
   e = false;
durch = null;
nichtEnthalten=v;
  }

  public FVariable(String name, Variable v) {
        this.name=name;
        e = false;
        durch = null;
        nichtEnthalten=v;
  }

  public Formel ersetzen(Formel f) {
    e = true;
    durch = f;
    return f;
  }

  public Formel getErsetzung() {
    return durch;
  }

  public String toString() {
```

```
    return name;
  }

  public String toString2() {
    if (nichtEnthalten==null) return name;
    else return name+" nv "+nichtEnthalten.
    toString2();
  }

  public boolean ersetzt() {
    return e;
  }

  public Variable getNichtEnthalten() {
   return nichtEnthalten;
  }

  public void setNichtEnthalten(Variable v) {
   nichtEnthalten=v;
  }


  /**
   * Gibt die Kosten einer Formelvariable zurück
   * @return Kosten der Formelvariable
   */
  public int laenge(){
   return laengenwert;
  }

  /**
   * Hier können die Kosten, die eine FVariable
   bei der Längenberechnung
   * bekommen soll angegeben werden.
   * @param Kosten, die eine FVariable bekommen soll
   */
  public static void setlaengenwert(int i) {
   laengenwert = i;
  }
}
**************************************************

package parser;
```

```
import java.util.Vector;
import java.io.*;
import java.lang.*;
import markierung.*;
import java.awt.Component;
import gui.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

// die Klasse Kb repräsentiert eine
Prädikatenlogische Wissensbasis

public class Kb extends AbstractTableModel
{
  String name;
  String description;

  private Vector  formula;
  private Vector  sortula;
  private Vector varibula;

  private Vector labelIcon;

  Logik allowedLogik[];
  Logik usedLogik;

/* Table Model Attribute ************************
  private int col = 3; //Anzahl der Zeile: 0 -# 1
  -Icon, 2 -Formel
  private int row;



  public void save(String dateiname)  // speichert
   die Kb unter dem Namen dateiname !
  {
   System.out.println("speichere KB -" + name +
   "- unter dem Namen " + dateiname + "!");

    Vector ausgabe = new Vector();
    String sortenzeile;

    ausgabe.add((String)"Beschreibung");
```

```
ausgabe.add((String)description);
ausgabe.add((String)"Individuenbereich");

for (int i = 0; i < sortula.size(); i++)
{
  Sorte help = (Sorte)sortula.get(i);
  sortenzeile = help.getName() + ":" +
  help.getIndividuenString();
  ausgabe.add((String)sortenzeile);

}
ausgabe.add((String)"Sortenteil");

String sh ="";
for (int i = 0;i<varibula.size();i++)
{ Term help = (Term)varibula.get(i);

  if(help instanceof Funktion) sh = "func ";
  if(help instanceof Konstante) sh = "const ";
  if(help instanceof Variable) sh = "var ";

  sh = sh + varibula.get(i) + " " +
  help.getSorte().getName();
  ausgabe.add((String)sh);
  sh = "";
}
ausgabe.add((String)"Formelteil");

sh = ""; //hilfsvariable
MarkFormel mf = null;
FLabel l;
for(int i = 0; i< formula.size(); i++)
{
  mf = (MarkFormel)formula.get(i);
  sh = mf.getFormel().toString();
  l = mf.getFLabel();
  sh = sh + " $ " + l.getString();
  ausgabe.add((String)sh);
}


try {

BufferedWriter out = new BufferedWriter(new
```

```
    FileWriter(dateiname));

   for (int i = 0; i < ausgabe.size(); i++) {
     out.write( (String) ausgabe.get(i)); //+ "\n");
     out.newLine();
   }

   out.close();
   }

   catch(Exception e)
   {
     System.out.println("Fehler beim schreiben
     der Datei aufgetreten !!!");
   }
}
```

```
public Kb(Vector v)
{
  this.formula = v;
  createIcons();
}

// woa der ultimative Konstruktor:
// v,w,x sind die 3 von Lader erzeugten
Vektoren für die Wissensbasis.

public Kb (String name, Vector v, Vector w,
 Vector x, String description)
{
 this.name = name;
 this.description = description;
  formula  = v;
  sortula  = w;
  varibula = x;
  createIcons();
}


public Kb (Vector v, Vector w, Vector x)
{
```

```
    formula  = v;
    sortula  = w;
    varibula = x;
    createIcons();
  }


  public Kb (String n, String d) {
    name = n;
    description = d;
  }

  public int getLength() {return formula.size();}

  public Vector getFormeln() {return formula;}
  public void setFormeln(Vector v) {this.formula = v;}

  public Vector getSorten() {return sortula;}
  public void setSorten(Vector v) {this.sortula = v;}

  public Vector getVariablen() {return varibula;}
  public void setVariablen(Vector v) {this.varibula = v;}

  public Vector getIcons() {return labelIcon;}
  public void setIcons(Vector v) {this.labelIcon = v;}


  public String getName() {
    return name;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String s) {
    description = s;
  }


  public String toString()
  {
    String ret = "";
    for(int i = 0; i< formula.size(); i++) ret
```

```
     = ret + formula.get(i).toString() + "\n";
    return ret;
  }

  public void output()
  {
    System.out.println("Die Wissensbasis: ===");
    System.out.println(this.toString());

    System.out.println("Die Terme der Wissensbasis
     sind:");
    for (int i = 0;i<varibula.size();i++)
    { Term help = (Term)varibula.get(i);
      System.out.println(varibula.get(i) + " " +
      help.getSorte().getName());
    }

    System.out.println("\nDie Sorten:");
    System.out.println(sortula);
  }

  public void createIcons() {
    if (labelIcon == null) labelIcon = new
    Vector(100, 100);
    for (int i = 0; i < formula.size(); i++)
      labelIcon.add(new LabelIcon( ( (MarkFormel)
      formula.get(i)).getFLabel(), 20, 0));
  }

/* Table Model Methoden ****************************/

  public int getRowCount() {return formula.size();}
  public int getColumnCount() {return col;}

  public String getColumnName(int columnIndex){
    // return columnIndex == 0 ? "Label" : "Formel";
    if (columnIndex == 0) return "Nummer";
    else if (columnIndex == 1) return "Label";
    else return "Formel";
  }
  public Class getColumnClass(int columnIndex) {
    if (columnIndex == 0)
      return String.class;
    else if (columnIndex == 1)
```

```
      return Icon.class;
    else return String.class;
  }

  public boolean isCellEditable(int rowIndex,
  int columnIndex) {
    return rowIndex < row && columnIndex < col;
  }
  public Object getValueAt(int rowIndex, int
   columnIndex) {
    MarkFormel mformel = (MarkFormel) formula.
    elementAt(rowIndex);
    String formel = Parser.
    sonderzeichenErsetzungsmethode(mformel.
    getFormel().toString());
    if (columnIndex == 0)
      return java.lang.Integer.toString(rowIndex);
    else if (columnIndex == 1)
      return labelIcon.get(rowIndex);
    else
      return formel;
  }
    public void setValueAt(Object aValue,
     int rowIndex, int columnIndex) {
      if (columnIndex == 0) {
 //       FLabel neu = (FLabel) aValue;
 //       MarkFormel mformel = (MarkFormel)
  fm.getFormeln().elementAt(rowIndex);
 //       mformel.setFLabel(neu);
      } else {
 //       Formel neu = (Formel) aValue;
 //       MarkFormel mformel = (MarkFormel)
 fm.getFormeln().elementAt(rowIndex);
  //      mformel.setFormel(neu);
      }
    }




}
*******************************************

package parser;
```

```
import parser.*;

// Steht für Konjungierte Formel:macht aus zwei
Formeln und einem binärem Operator
// eine dritte Formel
public class KonjFormel
    extends Formel {

  Formel right;
  Formel left;
  String op;
  static int laengenwert = 1;

  public KonjFormel(Formel links, Formel rechts,
  String operator) {
    right = rechts;
    left = links;
    op = operator;

  }

  public Object clone() {
    KonjFormel t = new KonjFormel(left, right, op);

    return t;
  }

  public void setRight(Formel neueformel) {
    this.right = neueformel;
  }

  public void setLeft(Formel neueformel) {
    this.left = neueformel;
  }

  public void setOperator(String neuerOp) {
    this.op = neuerOp;
  }

  public Formel getRight() {
    return this.right;
  }
```

```
  public Formel getLeft() {
    return this.left;
  }

  public String getOperator() {
    return this.op;
  }

  public String toString() {
    return "(" + left.toString() + " " +
    op + " " + right.toString() + ")";
  }

  public String toString2() {
    return "(" + left.toString2() + op +
    right.toString2() + ")";
  }

  public int laenge() {
   return right.laenge() + left.laenge() +
   laengenwert;
  }

  public static void setlaengenwert(int i) {
   laengenwert = i;
  }

}
***********************************************

/* Generated by Together */
package parser;

// eine prädikatenlogische Konstante
public class Konstante
    extends Term {
  private String name;

  public Konstante(String tanga, Sorte sorte) {
    name = tanga;
    super.sorte = sorte;
  }

  public String getName() {
```

```
    return name;
  }

  public Object clone() {
    return new Konstante(name, sorte);
  }

  public String toString() {
    return name;
  }

  public String toString2() {
    return name;
  }

}
*****************************************

package parser;

import java.util.*;
import java.io.*;
import java.lang.*;
import markierung.*;

public class MarkFormel {

  Formel formel;
  FLabel flabel;

  //*** der Konstruktor: ließt die Formel
   sowie die Label angaben ein
   // macht also aus dem String bestehend aus der
   Formel- und dem Lablel Teil ein
   // MarkFormel-Objekt
   public MarkFormel() {

   }

   public MarkFormel(String zeile) {
    flabel = new FLabel();
    StringTokenizer st = new StringTokenizer(zeile,
     "$");
    String test = new String(st.nextToken());
```

```
    formel = Parser.parse(test);
    String flabelStr = st.nextToken();
    StringTokenizer st2 = new StringTokenizer(
    flabelStr, ",");

    while (st2.hasMoreTokens()) {
      String stuetzStr = st2.nextToken();
      StringTokenizer st3 = new StringTokenizer
      (stuetzStr);
      String h1 = st3.nextToken();
      String h2 = st3.nextToken();
      //System.out.println(h1);
      //System.out.println(h2);
      flabel.add(new StPkt(java.lang.Double.
      parseDouble(h1),
                           java.lang.Double.
                           parseDouble(h2)));
    }
  }

  // ein zweiter Konstruktor: Die Formel und das
  Label werden direkt eingelesen
  public MarkFormel(Formel formel, FLabel label) {
    this.formel = formel;
    this.flabel = label;
  }

  public void setFormel(Formel neu) {
    this.formel = neu;
  }

  public void setFLabel(FLabel neu) {
    this.flabel = neu;
  }

  public Formel getFormel() {
    return this.formel;
  }

  public FLabel getFLabel() {
    return this.flabel;
  }

  public String toString() {
```

```
    if (formel == null) {
    System.out.println("MarkFormel.formel=null");
    }
    return new String("F: " + formel.toString()
    + " L: " + flabel.toString());
  }
  public String toString2() {
    if (formel == null) {
      System.out.println("MarkFormel.formel=null");
    }
    return new String("F: " + formel.toString2() +
    " L: " + flabel.toString());
  }

  public int laenge() {
return formel.laenge();
  }
}
*******************************************

package parser;

import parser.*;

//eine negierte Formel

public class NegFormel
    extends Formel {

  Formel formel;
  String op;
  static int laengenwert = 1;

  public NegFormel(Formel formel, String operator) {
    this.formel = formel;
    op = operator;
  }

  public void setFormel(Formel neueformel) {
    this.formel = neueformel;
  }

  public void setOperator(String neuerOp) {
    this.op = neuerOp;
```

```
  }

  public Formel getFormel() {
    return this.formel;
  }

  public String getOperator() {
    return this.op;
  }

  public String toString() {
    return op + "(" + formel.toString() + ")";
  }

  public String toString2() {
    return op + "(" + formel.toString2() + ")";
  }

  public int laenge() {
return formel.laenge() + laengenwert;
  }

  public static void setlaengenwert(int i) {
   laengenwert = i;
  }

}
*****************************************

package parser;

import java.util.*;
import java.io.*;
import java.lang.*;


// die Klasse Prädikat repräsentiert ein Prädikat


public class Praedikat
    extends Formel {
  private int arity;
  private Term[] terme;
  private String name;
```

```
static int laengenwert = 1;

//wird hier null als Term[] abgegeben, so
wird ein nullstelliges Prädikat,
// eine Aussagenvariable, erzeugt !!!

public Praedikat(String name, Term[] terme) {
  if (terme == null) {
    arity = 0;
  }
  else {
    arity = terme.length;
  }
  this.name = name;
  this.terme = terme;
}

public Praedikat(Term term, String name) {
 // ein 2. Konstruktor für nur einen Term
  if (term == null) {
    arity = 0;
  }
  else {
    arity = 1;
  }
  this.name = name;
  this.terme = new Term[1];
  this.terme[0] = term;
}

public Praedikat(String name, Term term1,
Term term2) { // ein 3. Konstruktor für 2 Terme
  arity = 2;
  this.name = name;
  this.terme = new Term[2];
  this.terme[0] = term1;
  this.terme[1] = term2;
}

public Term getTermNr(int i) {

  if (arity == 0) {
    return null; // im Falle einer Aussagenvariable
  }
```

```
    i--;
    if ( (i < 0) || (i >= arity)) {
      return null;
    }
    else {
      return terme[i];
    }
  }

  public void setTermNr(int i,Term t) {

    i--;
    if ( (i < 0) || (i >= arity)) {
      ;
    }
    else {
      terme[i] = t;
    }
  }

  public int getArity() {
    return arity;
  }

  public String toString() {
    if (arity == 0) {
      return name;
    }
    String ret;
    ret = name + "(";

    for (int i = 1; i < arity + 1; i++) {
      ret = ret + getTermNr(i).toString();
      if (i != arity) {
        ret = ret + ",";
      }
    }
    return ret + ")";
  }

  public String toString2() {
    if (arity == 0) {
      return name;
    }
```

```
    String ret;
    ret = name + "(";

    for (int i = 1; i < arity + 1; i++) {
      ret = ret + getTermNr(i).toString2();
      if (i != arity) {
        ret = ret + ",";
      }
    }
    return ret + ")";
  }

  public String getName() {
    return this.name;
  }

  public int laenge() {
return laengenwert;
  }

  public static void setlaengenwert(int i) {
   laengenwert = i;
  }
}
*******************************************

package parser;

import java.util.*;
import java.io.*;
import markierung.*;
import verarbeitung.*;

public abstract class Quantor implements
Serializable {

  public abstract FLabel erfuellt(Formel f,
  Vector kb) throws GefundenException;
  public abstract boolean gleicherQuantor
  (Quantor q);
  public abstract String toString();
}

package parser;
```

```
import java.util.*;
import markierung.*;
import verarbeitung.*;

// repräsentiert eine quantorisierte Formel

public class QuantorFormel extends Formel {
  private Formel wirkungsbereich;
  private String quantor;
  private Variable v;
  private Quantor q  = null;
  static int laengenwert = 1;

  public QuantorFormel(String quantor,
  Variable v, Formel f) {
    this.quantor = quantor;
    this.v = v;
    this.wirkungsbereich = f;
    if(quantor.equals("@"))q=new Allquantor();
     //das logische "für alle"
    if(quantor.equals("?"))q=new Existenzquantor();
    // das logische "existiert"

  }
  //schafft einen Quantorisierte Formel
  public QuantorFormel(Quantor quantor,
  Variable v, Formel f) {
    this.q = quantor;
    this.quantor = q.toString();
    this.v = v;
    this.wirkungsbereich = f;
  }

  public Formel getFormel() {
    return wirkungsbereich;
  }

  public Quantor getQuantor() {
    return q;
  }
  public Variable getVariable() {
    return v;
```

```
  }

  public void setVariable(Variable var) {
   v=var;
  }

  public void setFormel(Formel f) {
    wirkungsbereich = f;
  }

  public FLabel erfuellt(Formel f, Vector
   kb) throws GefundenException {
     return q.erfuellt(f, kb);
  }

  public String toString() {
   return "("+quantor + v + ") " + wirkungsbereich;
  }

  public String toString2() {
   return quantor + v.toString2() + " " +
   wirkungsbereich.toString2();
  }

  public int laenge() {
return wirkungsbereich.laenge() + laengenwert;
  }

  public static void setlaengenwert(int i) {
   laengenwert = i;
  }
}
**********************************************

/* Generated by Together */

package parser;

import parser.*;
import java.io.*;

// Repräsentation einer prädikatenlogischen Sorte
// die KLasse Sorte enhält ein Array mit der Liste
der gültigen Variablen,
```

```
// also quasi den Individuenbereich der Sorte.

public class Sorte implements Serializable {

  String name;
  String[] individuenbereich = {"Hallo"};
  Konstante indibereich[];

  // der Individuenbereich wird hier als
  String übergeben
  public Sorte(String name, String[]
  individuenbereich) {
    this.name = name;
    this.individuenbereich = individuenbereich;

    indibereich = new Konstante[individuenbereich.
    length];


    for(int i = 0; i<individuenbereich.length;i++)
    {
      indibereich[i] = new Konstante
      (individuenbereich[i],this);
    }

  }

  public Sorte(String name) {
    this.name = name;
  }

  // gibt zurück, oder der String s im
  Individuenbereich der Sorte liegt !!
  public boolean IstTeilvon(String s) {
    boolean ret = false;
    for (int i = 0; i < individuenbereich.length; i++) {
      if (individuenbereich[i].equals(s)) {
        ret = true;
      }
    }
    return ret;
  }
```

```
  //ist die Konstante k teil des individuenbereichs ?
  public boolean IstTeilvon(Konstante k)
  {
   return IstTeilvon(k.getName());
  }


  public String[] getElements() {
    return individuenbereich;
  }

  public Konstante[] getKonstanten() {
  //System.out.println("test");
  //System.out.println(individuenbereich);
  Konstante[] k = new Konstante[individuenbereich.
  length];
   for(int i=0; i<individuenbereich.length; i++) {
k[i]=new Konstante(individuenbereich[i], this);
}
return k;
  }

  //liefert den individuenbereich dargestellt
  als Array von Konstanten !
  public Konstante[] getKonst() // diese
  Methode sollte benutzt werden !!!!
  {
   return indibereich;
  }

  public void setIndividuenBereich (String[] s){
    this.individuenbereich = s;

    indibereich = new Konstante[s.length];


    for(int i = 0; i<s.length;i++)
    {
      indibereich[i] = new Konstante(s[i],this);
    }

  }
```

```
  public String getName() {
    return name;
  }

  public void setName(String neuerName) {
    this.name = neuerName;
  }

  public String toString() {
    String ret;
    ret = "Name der Sorte: " + name +
     "\n Individuenbereich: ";
    if (individuenbereich != null) {
      for (int i = 0; i <
      individuenbereich.length; i++) {
        ret = ret + individuenbereich[i] + ", ";
      }
    }
    return ret + "\n";
  }

 // eine Hilfsmethode
 public String getIndividuenString()
 {
   String ret = "";
   for (int i = 0; i < individuenbereich.length;
    i++)
   {
        ret = ret + individuenbereich[i];
        if (i!= individuenbereich.length-1)
        ret = ret + " ";
   }

   return ret;
 }
}

*****************************************

/* Generated by Together */
package parser;

import java.io.*;
```

```
// diese Klasse repräsentiert einen
prädikatenlogischen Term

abstract public class Term implements
Serializable {
  String name;
  Sorte sorte;

  public String getName() {
    return name;
  }

  public Sorte getSorte() {
    return sorte;
  }

  public void setSorte(Sorte sorte) {
    this.sorte = sorte;
  }

  abstract public String toString2();

}

************************************************

/* Generated by Together */

/**
 * Individuenvariable
 */

package parser;

import java.io.*;

// die Klasse Variable repräsentiert einen
Prädikatenlogische Individuenvariable
//
public class Variable
    extends Term {

  // schafft einen Variable mit Namen "tanga"
```

```
  und Sorte "sorte"
  public Variable(String tanga, Sorte sorte) {
    name = tanga;
    super.sorte = sorte;
//System.out.println("neue Variablenzuordnung:
" +tanga+ " -> " +sorte);
  }

  public void setName(String neuerName) {
    name = neuerName;
  }

    /*
    public Object clone() {
    return new Variable(name, sorte);
  }
    */


  public String toString() {
    return name;
  }


  public String toString2() {
    return name+"#"+hashCode()+"#";
  }

}

*******************************************
package parser;

import java.util.Vector;
import markierung.*;
import verarbeitung.*;

public class Allquantor
    extends Quantor {

private boolean d=false;

    public Allquantor(){
;
```

```
    }

    public String toString(){
return "@";
    }

/**
     * Diese Methode wird bei der Überprüfung
     von Allquantoren verwendet.
 * Übergeben wird eine konjugierte Formel
 (Implikation), welche auf der linken Seite als
  Prämisse eine quantorisierte Formel enthält.
 * In dieser quantorisierten Formel werden für
  die vom Quantor abhängige Variable sukzessive
  alle Individuen der zugehörigen Sorte eingesetzt
 * und überprüft, ob diese Formel in der Wissensbasis
 vorhanden ist. Um den Allquantor zu erfüllen müssen
  alle Individuen ein positives Resultat liefern.
 * Dies enspricht einer Konjunktion über alle
 Individuen eingesetzt in die quantorisierte Formel.
 * Die Bewertung der quantorisierten Formel
 ergibt sich aus dem FLabel des "schlechtesten"
 Faktors bei der Einsetzung. Diese Bewertung wird
  zurückgegeben
 * und dient als Bewertung für die Konklusion
 der übergebenen Implikation.
 * @param fcopy Implikation, welche auf der
 linken Seite eine quantorisierte Formel enthält.
 * @param kb Wissensbasis, welche auf jeden
  einzelnen Faktor überprüft wird.
 * @return gibt ein FLabel zurück, welches
 als Bewertung für die Konklusion der übergebenen
  Implikation dient.
 * @exception GefundenException nur Weiterleitung
 der Exception
    */

   public FLabel erfuellt(Formel fcopy, Vector kb)
    throws GefundenException{
      //Allquantor->Konjunktion über alle
      Individuen der Sorte
      QuantorFormel qf=(QuantorFormel) (
      (KonjFormel)fcopy).getLeft();
      boolean erfuellt = false;
```

```
      Variable va = qf.getVariable();
      Sorte so = va.getSorte();
      Konstante[] individuen;
      individuen = so.getKonstanten();
      FLabel temp = new FLabel();
      //Beim Allquantor beginnt man mit
      absolute_true und ersetzt immer dann
       wenn schlechtere Bewertungen gefunden wurden
      temp.set_AbsTrue();

      if (d) System.out.println("Allquantor "+qf);
  //Der Wirkungsbereich des Quantors wird für
   alle Individuen überprüft
      for (int i=0;i<individuen.length;i++)
      {
       if (d) System.out.println("einsetzen von
        "+individuen[i]);

        QuantorFormel temptesting = (QuantorFormel)
        InfTool.kopiere(qf);  //kopieren
    Variable gesucht = temptesting.getVariable();
     //gebundene Variable bestimmen
        Formel testing = temptesting.getFormel();
        InfTool.varDurchKonstErsetzen(testing,

         gesucht, individuen[i]); //in Kopie einsetzen
        MarkFormel bewFormel = InfTool.suchInKB_
        q(testing, fcopy); //in KB suchen
   qf=(QuantorFormel) ((KonjFormel)fcopy).getLeft();

   if ((bewFormel == null) && (testing
   instanceof KonjFormel)) bewFormel = InfTool.
   suchInKBund(testing);  //andere Möglichkeit
   zu suchen

        if(bewFormel == null){
       erfuellt = false; //schon bei einem
       Fehlschlag wird abgebrochen und die Formel
        gilt als nicht erfüllt.
       break;
     }
         else {
if (d) System.out.println("sieht gut aus ");
       temp = temp.mergeMin(bewFormel.getFLabel());
```

```
       //solange Formeln gefunden werden,
        wird sich die schlechteste Bewertung gemerkt.
       erfuellt = true;
          }
      }
      if(erfuellt)return temp;
      else return null;
  }

  public boolean gleicherQuantor(Quantor q) {
   if (q instanceof Allquantor) return true;
else return false;
  }

}

*****************************************


package parser;

import java.util.Vector;

// eine Klasse die eine Logik repräsentiert

public class Logik
{
  Vector v; // für die Axiome
  String name;


  public Logik(Vector v)
  {
   this.v = v;

  }

  public Vector getVector() {return v;};
  public void setVector(Vector v) {this.v = v;};

  public String toString()
  {
    String ret = "";
    for(int i = 0; i< v.size(); i++) ret = ret +
```

```
     v.get(i).toString() + "\n";
    return ret;
  }
}

*****************************************

package parser;


// hallo Gregor: hab hier Änderungen bezüglich
der Axiome vorgenommen:
// hab den axiomVector hinzugefügt und die
Methode ladeAxiome eingefügt
// die Methode ist allerding noch
auskommentiert (siehe unten)
// Thomas

import java.util.*;
import java.io.*;
import java.lang.*;

public class Lader {

  public static Vector individuenVector
   = new Vector(100, 100);
  public static Vector sortenVector =
  new Vector(100, 100);
  public static Vector formelVector =
  new Vector(100, 100);
  public static Vector axiomVector =
  new Vector(100,100);
  static String fName, nFreiVar; // Werden für die
  nicht freien Variablen bei Axiomen benötigt

  // Hi Gregor hab mir mal erlaubt hier eine
   Methode einzufügen (thomas):

  public static Kb getKb(String filename,
  String name) // liefert eine Wissensbasis zurück
  {
    Vector sorten = ladeIndividuenbereich(filename);
    // Sorten mit Individuen
    Vector terme  = ladeSorten(filename);
```

```
    //notwendig, damit der Parser die Sorten
    setzen kann!!
   Vector formeln = Lader.ladeFormeln(filename);
   String demoDescription = Lader.
   ladeKbBeschreibung(filename);
   String demoKbName = name;

   return new Kb(demoKbName,formeln,
   sorten,terme,demoDescription);
 }


 public static String kommentarEntfernen
 (String eineZeile) {

   eineZeile = eineZeile.trim(); // Leerzeichen
    am Anfang & Ende der Zeile entfernen

   if (eineZeile.startsWith("//")) {
     return "";
   } //Beginnt die Zeile mit einem Kommentar
    --> leere Zeile zurückgeben

   if (eineZeile.indexOf("//") != ( -1)) {
   //Kommentar an anderer Stelle vorhanden?
     String[] splitKommentar = eineZeile.split("//");
      // Zeile an Kommentarmarkierung aufspalten
     return splitKommentar[0]; // Zeile ohne
      Kommentar zurückgeben
   }

   else {
     return eineZeile; // falls kein Kommentar
      --> ursprüngliche Zeile zurückgeben
   }
 }

 public static Vector getIndividuenVector() {
  return individuenVector;
 }

 /* Überprüft, ob im individuenVector eine
 Sorte mit dem Namen "sortenName" vorhanden
  *  ist. Falls ja, wird das entsprechende
```

```
    Element  zurückgegeben.
   *  Falls nein, wird "-1" zurückgegeben, was
   bei der aufrufenden Funktion eine Exception
   *  auslösen kann.
   */

   public static String ladeKbBeschreibung(String
   filename) {

String beschreibung="";

          try {
            BufferedReader in = new BufferedReader(new
            FileReader(filename));
            String bZeile;
            boolean descriptionBorder = false;

            while (true) {
              bZeile = in.readLine();

              if (bZeile == null) {
                break;
              }

              bZeile = kommentarEntfernen(bZeile);
              if ( (descriptionBorder == true)) {

                if (bZeile.equals("Individuenbereich")) {
                  break;
                }

                beschreibung = beschreibung.concat(bZeile);

              }

              if (bZeile.equals("Beschreibung")) {
                descriptionBorder = true;
              }
            }

          }
          catch (Exception e) {
            System.out.println(
                "Datei nicht gefunden oder keine
```

```
                Beschreibung definiert!");
          }

    return beschreibung;
  }



  public static Sorte sorteImVectorFinden(String
   sortenName) {
    int sortenPos = -1;
//System.out.println(sortenName);
    for (int i = 0; i < individuenVector.size();
    i++) {
      Sorte aktuelleSorte = (Sorte)
      individuenVector.elementAt(i);

      if (aktuelleSorte.name.equals(sortenName)) {
        sortenPos = i;
      }
    }

    return (Sorte) individuenVector.get(sortenPos);
  }

  public static Vector ladeIndividuenbereich(String
  filename) {

      try {
        BufferedReader in = new BufferedReader(new
        ileReader(filename));
        String individZeile;
        boolean individBorder = false;
        while (true) {
          individZeile = in.readLine();
          //System.out.println("Die Zeile lautet:
          "+individZeile);

          if (individZeile == null) {
            break;
          }

          individZeile = kommentarEntfernen(individZeile)
           // vorhandenen Kommentar entfernen
```

```
        //System.out.println("Die Zeile ohne Kommentar
         lautet: "+individZeile);
        //if (individZeile != "") {
        if ( (individBorder == true) && (individZeile
        != "")) {

          if (individZeile.equals("Sortenteil")) {
            break;
          }

          //Es wird nun die Sorte vom Individuenbereich
          am ":" abgetrennt
          String[] splitIndivid = individZeile.
          split(":");
          //String[] individuenteil =
          splitIndivid[1].split(" ");
          //System.out.println(splitIndivid[1]);
          //System.out.println("Länge des
          individuenteil ist: " + individuenteil.length);
          //System.out.println("Die Arraygroesse
           im Individuenteil ist: " +splitIndivid.length);

          //Der Individuenbereich wird ebenfalls
          aufgeteilt & damit ein neues Sortenobjekt erzeugt
          Sorte neueSorte = new Sorte(splitIndivid[0],
                                      (splitIndivid[1].
                                      trim()).split(" "));
          individuenVector.add(neueSorte);
        }

        if (individZeile.equals("Individuenbereich")) {
          individBorder = true;
        }
      }

    }

    catch (Exception e) {
      individuenVector.removeAllElements();
      System.out.println("Datei nicht gefunden
      oder Wissensbasis im Individuenbereich
      entspricht nicht den Konventionen");
    }
  return individuenVector;
```

```
}



public static Vector ladeSorten(String filename) {

  try {
    BufferedReader in = new BufferedReader(new
     FileReader(filename));
    String tempZeile;
    boolean formelBorder = false;

    while (true) {
      tempZeile = in.readLine();
      if (tempZeile == null) {
        break;
      }
      tempZeile = kommentarEntfernen(tempZeile);

      if (formelBorder == true) {

        if (tempZeile.equals("Formelteil")) {
          break;
        }

        String[] splitZeile = tempZeile.split("
         "); //eingelesene Zeile an Leerzeichen aufsplitten

        try {
          if (splitZeile[0].equals("var")) {
          //aufgesplitte Zeile wird als Array behandelt

            String name = splitZeile[1];

            Sorte sorte = sorteImVectorFinden
            (splitZeile[2]);
            Variable sorteVar = new Variable
            (name, sorte);
            sortenVector.add(sorteVar); //
             Sorte Variable dem Vector hinzufügen

          }
        }
        catch (Exception e) {
```

```
  System.out.println(
      "Es wurden noch keine
       Individuenbereiche in den
        IndividuenVector geladen");
}

try {
  if (splitZeile[0].equals("const")) {

    String name = splitZeile[1];
    Sorte sorte = sorteImVectorFinden
    (splitZeile[2]);
    Konstante sorteKonst = new Konstante
    (name, sorte);
    sortenVector.add(sorteKonst);

  }
}
catch (Exception e) {
  System.out.println(
      "Es wurden noch keine Individuenbereiche

      in den IndividuenVector geladen");
}

try {

  if (splitZeile[0].equals("func")) {

    String name = splitZeile[1];
    Sorte sorte = sorteImVectorFinden
    (splitZeile[splitZeile.length -
                                    1]);

    Term[] term = new Term[splitZeile.length - 3];

    for (int i = 0; i < term.length; i++) {

      for (int v = 0; v < sortenVector.size();
       v++) {
        String stringname = sortenVector.
        elementAt(v).toString();
        if (stringname.equals(splitZeile[i
        + 2])) {
```

```
                if ( (sortenVector.elementAt
                (v)instanceof Variable) ||
                    (sortenVector.elementAt(v)
                    instanceof Konstante) ||
                    (sortenVector.elementAt(v)
                    instanceof Funktion)) {

                  term[i] = (Term) sortenVector.
                  elementAt(v);
                }
                else {
                  System.out.println(
                      "Fehler: Funktion benutzt
                      bisher nicht definierte
                      Parameter");

                }
              }

            }

          }

          Funktion sorteFunk = new Funktion(name,
          term, sorte);
          sortenVector.add(sorteFunk);

        }
      }
      catch (Exception e) {
        System.out.println(
            "Es wurden noch keine Individuenbereiche
             in den IndividuenVector geladen");
      }
    }

    if (tempZeile.equals("Sortenteil")) {
      formelBorder = true;
    }
  }
}

catch (Exception e) {
```

```
    System.out.println("Datei nicht vorhanden! " + e);
  }

  return sortenVector;
}

/*
 *    Noch auskommentiert, da die Methode parser
 .parseAxiom(String) noch
 *    nicht implementiert ist.
 *    Die Methode ladeAxiome wurde deshalb noch
  nicht getestet, sollte aber
 *    funktionieren!!!
 *
 */
public static Vector ladeAxiome(String filename) {

    try {
      BufferedReader in = new BufferedReader(new
       FileReader(filename));
      String temp;
      boolean axiomBorder = false;

      while (true) {
        temp = in.readLine();
        if (temp == null) {
          break;
        }
        //System.out.println("Axiom mit Kommentar:
         "+temp);
        temp = kommentarEntfernen(temp);

        //System.out.println("Axiom ohne Kommentar:
        "+temp);

        if ( (axiomBorder == true) && (temp != "")) {

          if (temp.indexOf(":") < 0) {
            nFreiVar = null;
            fName = null;
          } // System.out.println("nix unfrei");}
          else { //System.out.println("Es gibt was
           zu tun!");
            String[] splitAxiom = temp.split(":");
```

```
              //Axiom am ":" aufspalten
              temp = splitAxiom[0].trim(); //
              unnötige Leerzeichen entfernen
              //System.out.println("Axiom ohne
              Restzeugs: "+temp);
              //System.out.println("Restzeugs :"
              + splitAxiom[1]);

              String[] splitAxiom2 = (splitAxiom[1].
              trim()).split(" "); // unnötige
              Leerzeichen entfernen

              nFreiVar = splitAxiom2[0];
              //System.out.println(nFreiVar);
              fName = splitAxiom2[1];
              //System.out.println(fName);
              //System.out.println(getNichtFrei(fName));
            }

            axiomVector.add(Parser.parseAxiom(temp));
             // die Axiome werden zu "Formeln" geparst

          }

          if (temp.equals("Axiomteil")) {
            axiomBorder = true;
          }

        }
      }

      catch (Exception e) {
        axiomVector.removeAllElements();
        System.out.println("Axiom-Datei
        nicht vorhanden! " + e);
        e.printStackTrace();
      }
    return axiomVector;
  }

   public static String getNichtFrei(String
   name){

    if (name.equals(fName)) {return nFreiVar;}
```

```
  //Gibt die nicht freie Variable zurück
  return null;
  //falls gesetzt
 }


public static Vector ladeFormeln(String
filename) {

    try {
      BufferedReader in = new
      BufferedReader(new FileReader(filename));
      String temp;
      boolean border = false;
      while (true) {
        temp = in.readLine();

        if (temp == null) {
          //System.out.println("null");
          break;
        }
        temp = kommentarEntfernen(temp);

        if ( (border == true) && (temp != "")) {
          MarkFormel neueMF = new MarkFormel(temp);
           // das parsen geschieht in MarkFormel
          formelVector.add(neueMF);
        }

        if (temp.equals("Formelteil")) {
          border = true;
        }
      }
    }

    catch (Exception e) {
      formelVector.removeAllElements();
      System.out.println("Datei nicht vorhanden! ");
      e.printStackTrace();
    }
  return formelVector;
}
```

```
public static void removeAll() {
  formelVector.removeAllElements();
  sortenVector.removeAllElements();
  individuenVector.removeAllElements();
  axiomVector.removeAllElements();
}




boolean isVariable(String s) {
  boolean vorhanden = false;

  //Vector durchsuchen. Falls Element mit
   gefordertem String
  //vorhanden ist: gucken ob entsprechendes
  Element eine Instanz von Variable ist

  for (int i = 0; i < sortenVector.size(); i++) {

    String name = sortenVector.elementAt(i)
    .toString();
    if ( (name.equals(s)) && (sortenVector.
    elementAt(i)instanceof Variable)) {
      vorhanden = true;
    }

  }
  return vorhanden;
}

boolean isKonstante(String s) {
  boolean vorhanden = false;

  //Vector durchsuchen. Falls Element mit
  gefordertem String
  //vorhanden ist: gucken ob entsprechendes
  Element eine Instanz von Konstante ist

  for (int i = 0; i < sortenVector.size();
   i++) {

    String name = sortenVector.elementAt(i).
```

```
      toString();
      if ( (name.equals(s)) && (sortenVector.
      elementAt(i)instanceof Konstante)) {
        vorhanden = true;
      }

    }
    return vorhanden;
  }

  public boolean isFunktion(String s) {
    boolean vorhanden = false;

    //Vector durchsuchen. Falls Element mit
     gefordertem String
    //vorhanden ist: gucken ob entsprechendes
    Element eine Instanz von Konstante ist

    for (int i = 0; i < sortenVector.size(); i++) {

      String name = sortenVector.elementAt(i).
      toString();
      if ( (name.startsWith(s + "\u0028")) &&
          (sortenVector.elementAt(i)instanceof
          Funktion)) {
        vorhanden = true;
      }

    }
    return vorhanden;
  }

  //Durchsucht für einen String den sortenVector
   und liefert für das entsprechende
  //Element ein Sorten-Array zurück.
  public Sorte[] getSorten(String s) {

    int elementPosition = -1; // die Position
    an der Element mit Namen "s" zu finden ist
    Sorte[] welcheSorten = new Sorte[1];

    // feststellen, an welcher Position im Vector
    das Element ist
    for (int i = 0; i < sortenVector.size(); i++) {
```

```
  String name = sortenVector.elementAt(i).
  toString();

  if ( (name.equals(s) == true) ||
  name.startsWith(s + "\u0028"))) {
    elementPosition = i;
    break;
  }

}

// Fall Element eine Variable ist,
wird deren Sorte zurückgegeben
if (sortenVector.elementAt(elementPosition)
instanceof Variable) {
  welcheSorten[0] = ( (Variable) s
  ortenVector.elementAt(elementPosition)).
      sorte;
  return welcheSorten;

}

// Fall Element eine Konstante ist, wird deren
 Sorte zurückgegeben
if (sortenVector.elementAt(elementPosition)
instanceof Konstante) {
  welcheSorten[0] = ( (Konstante) sortenVector
  .elementAt(elementPosition)).
      sorte;
  return welcheSorten;

}

if (sortenVector.elementAt(elementPosition)
instanceof Funktion) {

  int groesse = ( (Funktion) sortenVector.
  elementAt(elementPosition)).
      getArity();
  int count = 0;

  Sorte[] funkSorten = new Sorte[groesse + 1];
  Funktion neueFunk = (Funktion) sortenVector.
```

```
    elementAt(elementPosition);
    funkSorten[groesse] = neueFunk.sorte;

    for (int t = 1; t <= groesse; t++) {
      String neu = (String) ( (Funktion)
      sortenVector.elementAt(
          elementPosition)).getTermNr(t)
          .toString();

      for (int g = 0; g < sortenVector.size();
      g++) {
        String neuertest = sortenVector.
        elementAt(g).toString();
        if (neuertest.startsWith(neu)) {

          if (isVariable(neuertest)) {
            Variable neueVar = (Variable)
            sortenVector.elementAt(g);
            funkSorten[count] = neueVar.sorte;
            count = count + 1;
          }

          if (isKonstante(neuertest)) {
            Konstante neueKonst = (Konstante)
             sortenVector.elementAt(g);
            funkSorten[count] = neueKonst.sorte;
            count = count + 1;
          }

          if (isFunktion(neuertest)) {
            Funktion neueFunk2 = (Funktion)
             sortenVector.elementAt(g);
            funkSorten[count] = neueFunk2.sorte;
            count = count + 1;

          }

        }
      }
    }

    return funkSorten;
  }
```

```
    return welcheSorten;
  }


   public static Kb loadKb (String name,
    String filename) {

     ladeAxiome("AxiomePL.txt");

/* usedLogic und allowedLogic müssten
noch gesetzt werden */

     return getKb(filename, name);
  }

}
```

9.4 Auszüge aus dem GUI-Quelltext

```
package gui;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import markierung.*;

public class LabelEditorPanel
    extends JComponent
    implements MouseInputListener, ListDataListener {

  private StPktList points;
  private int inset;
  private double gridRangeX;
  private double gridRangeY;
  private double gridResolution;
  private int accuracy;
  private Color graphColor;
  private Color gridColor;
  private Color backgroundColor;
  private int componentWidth;
```

```
private int componentHeight;
private String coords;
private Point compPoint;
private StPkt modelPoint;
private Rectangle boundRect;
private Rectangle pointRect;
private int pointListIndex;
private boolean dragged;

public LabelEditorPanel() {
  super();
  init();
  points = new StPktList();
  points.add(new StPkt(0.0, 0.0));
  points.add(new StPkt(gridRangeX, gridRangeY));
}

public LabelEditorPanel(StPktList s) {
  super();
  init();
  points = s;
}

public void setStPktList(StPktList s) {
  points = s;
  repaint();
}

public StPktList getStPktList() {
  return points;
}

public void paint(Graphics g) {
  int compx1;
  int compy1;
  int compx2;
  int compy2;

  super.paint(g);
  componentWidth = this.getWidth();
  componentHeight = this.getHeight();
  paintGrid(g);

  g.setColor(graphColor);
```

```
    if (points.size() > 1) {
      for (int i = 0; i < points.size() - 1;
      i++) {
        compx1 = getComponentX( ( (StPkt)
         points.get(i)).getT());
        compy1 = getComponentY( ( (StPkt)
         points.get(i)).getD());
        compx2 = getComponentX( ( (StPkt)
        points.get(i + 1)).getT());
        compy2 = getComponentY( ( (StPkt)
         points.get(i + 1)).getD());
        g.drawRect(compx1 - ( (int) pointRect.
        getWidth() / 2),
                   compy1 - ( (int) pointRect.
                   getHeight() / 2),
                   ( (int) pointRect.getWidth()),
                    ( (int) pointRect.getHeight()));
        g.drawRect(compx2 - ( (int) pointRect.
        getWidth() / 2),
                   compy2 - ( (int) pointRect.
                   getHeight() / 2),
                   ( (int) pointRect.getWidth()),
                   ( (int) pointRect.getHeight()));
        g.drawLine(compx1, compy1, compx2, compy2);
      }
    }
    else {
      if (points.size() == 1) {
        compx1 = getComponentX( ( (StPkt)
        points.get(1)).getT());
        compy1 = getComponentY( ( (StPkt)
         points.get(1)).getD());
        g.drawRect(compx1 - ( (int) pointRect.
        getWidth() / 2),
                   compy1 - ( (int) pointRect.
                   getHeight() / 2),
                   ( (int) pointRect.getWidth()),
                   ( (int) pointRect.getHeight()));
      }
    }
  }

  public void paintGrid(Graphics g) {
    double xc = 0.0;
```

```
    double yc = 0.0;
    int x1;
    int x2;
    int y1;
    int y2;
    int swidth = g.getFontMetrics().
    stringWidth(coords);

    g.setColor(backgroundColor);
    g.fillRect(0, 0, componentWidth,
    componentHeight);

    g.setColor(gridColor);
    y1 = componentHeight - inset;
    y2 = 0 + inset;
    while (xc <= gridRangeX) {
      x1 = (int) ( ( (xc / gridRangeX) *
      (componentWidth - 2 * inset)) + inset);
      g.drawLine(x1, y1, x1, y2);
      xc += gridResolution;
    }

    x1 = inset;
    x2 = componentWidth - inset;
    while (yc <= gridRangeY) {
      y1 = componentHeight -
          (int) ( ( (yc / gridRangeY) *
          (componentHeight - 2 * inset)) + inset);
      g.drawLine(x1, y1, x2, y1);
      yc += gridResolution;
    }

    g.drawString(coords, (componentWidth -
    swidth) / 2, inset - 3);

  }

  public void mouseClicked(MouseEvent e) {

  }

  public void mouseEntered(MouseEvent e) {
    if (checkGridBounds(e.getX(), e.getY()))
    {
```

```
    coords = "T " + getModelX(e.getX())
    + " D " + getModelY(e.getY());
    repaint();
  }
}

public void mouseExited(MouseEvent e) {
  coords = "";
  repaint();
}

public void mousePressed(MouseEvent e) {
  setMouseCoordinates(e);
  setPointListIndex(e);
}

public void mouseReleased(MouseEvent e) {
  if ( (!dragged)) {
    if ( (e.getButton() == e.BUTTON1) &&
    (checkGridBounds(e.getX(), e.getY())) &&
        pointListIndex == -1) {
      int i = 0;
      while ( (i < points.size()) &&
             ( (compPoint.getX() >
                getComponentX( ( (StPkt)
                points.get(i)).getT()))
              ||
              ( (compPoint.getX() ==
                 getComponentX( ( (StPkt)
                  points.get(i)).getT())) &&
               (compPoint.getY() >
                getComponentY( ( (StPkt)
                points.get(i)).getD())))))) {
        i++;

      }
      if ( (i > 0) && (i < points.size())) {
        setPointBounds(i - 1, i);

        if (checkBounds() == 3) {
          points.add(i, modelPoint);
          modelPoint = new StPkt(0.0, 0.0);
        }
      }
```

```
    }
    else if ( (e.getButton() == e.BUTTON3)
    && (pointListIndex > 0) &&
             (pointListIndex < points.size()
              - 1)) {
      points.remove(pointListIndex);
    }
  }
  else {
    dragged = false;
    pointListIndex = -1;
    points.fireModelChanged();
  }
}

public void mouseDragged(MouseEvent e) {
  int chk;
  double newx;
  double newy;
  dragged = true;

  if (pointListIndex > -1) {
    setMouseCoordinates(e);
    chk = checkBounds();
    if ( (pointListIndex > 0) &&
    (pointListIndex < points.size() - 1)) {
      if (chk == 3) {
        ( (StPkt) points.get(pointListIndex)).
        setLocation(modelPoint);
        coords = "T " + modelPoint.getT() + "
        D " + modelPoint.getD();
      }
      else if (chk == 2) {
        newy = getModelY( (int) (
        (compPoint.getY() < boundRect.getMinY()) ?
                                  boundRect.
                  getMinY() : boundRect.getMaxY()));
        ( (StPkt) points.get(pointListIndex)).
        setLocation(modelPoint.
            getT(), newy);
        coords = "T " + modelPoint.getT() + "
         D " + newy;
      }
      else if (chk == 1) {
```

```
          newx = getModelX( (int) ( (compPoint.getX()
          < boundRect.getMinX()) ?
                                    boundRect.getMinX()
                    : boundRect.getMaxX()));
          ( (StPkt) points.get(pointListIndex))
          .setLocation(newx,
              modelPoint.getD());
          coords = "T " + newx + " D " +
          modelPoint.getD();
        }
      }
      else {
        if ( (chk == 1) | (chk == 3)) {
          ( (StPkt) points.get(pointListIndex))
          .setLocation( (pointListIndex ==
              0) ? 0.0 : gridRangeX,
              odelPoint.getD());
          coords = "T " + ( (pointListIndex == 0)
           ? 0.0 : gridRangeX) + " D " +
              modelPoint.getD();
        }
        else {
          newy = getModelY( (int) (
          (compPoint.getY() < boundRect.getMinY()) ?
            boundRect.getMinY() : boundRect.getMaxY()));
          ( (StPkt) points.get(pointListIndex)).setLocation
          ( (pointListIndex ==
              0) ? 0.0 : gridRangeX, newy);
          coords = "T " + ( (pointListIndex == 0) ?
          0.0 : gridRangeX) + " D " +
              newy;
        }
      }
      repaint();
    }
  }

  public void mouseMoved(MouseEvent e) {
    if (checkGridBounds(e.getX(), e.getY())) {
      coords = "T " + getModelX(e.getX()) +
      " D " + getModelY(e.getY());
      repaint();
    }
  }
```

```
private void init() {
  inset = 20;
  gridRangeX = 1.0;
  gridRangeY = 1.0;
  gridResolution = 0.2;
  accuracy = 2;
  gridColor = new Color(200, 200, 200);
  graphColor = new Color(0, 0, 0);
  backgroundColor = new Color(255, 255, 255);
  coords = "";
  pointListIndex = -1;
  compPoint = new Point();
  modelPoint = new StPkt(0.0, 0.0);
  boundRect = new Rectangle();
  pointRect = new Rectangle(10, 10);
  dragged = false;
}

private boolean checkGridBounds(int x, int y) {
  if ( (x < inset) || (x > componentWidth -
  inset)
      || (y < inset) || (y > componentHeight
      - inset)) {
    return false;
  }
  else {
    return true;
  }
}

private int checkBounds() {
  if ( (compPoint.getX() >= boundRect.getMinX()) &&
      (compPoint.getX() <= boundRect.getMaxX())
      && (compPoint.getY() >= boundRect.getMinY()) &&
      (compPoint.getY() <= boundRect.getMaxY())) {
    return 3;
  }
  else if ( (compPoint.getX() >= boundRect.getMinX()) &&
           (compPoint.getX() <= boundRect.getMaxX())) {
    return 2;
  }
  else if ( (compPoint.getY() >= boundRect.getMinY()) &&
           (compPoint.getY() <= boundRect.getMaxY())) {
```

```
    return 1;
  }
  else {
    return 0;
  }
}

private void setPointListIndex(MouseEvent e) {
  int pointX;
  int pointY;

  pointListIndex = -1;

  for (int i = 0; i < points.size(); i++) {
    pointX = getComponentX( ( (StPkt) points.get(i)).
    getT());
    pointY = getComponentY( ( (StPkt) points.get(i)).
    getD());
    pointRect.setLocation(pointX - ( (int) pointRect.
    getWidth() / 2),
                          pointY - ( (int) pointRect.
                          getHeight() / 2));
    if ( (pointRect.contains(compPoint.getX(),
    compPoint.getY())) &&
        ( (i < points.size() - 1) ||
        (pointListIndex == -1))) {
      pointListIndex = i;
      setPointBounds(pointListIndex -
      1, pointListIndex + 1);
    }
  }
}

private void setMouseCoordinates(MouseEvent e) {
  compPoint.setLocation(e.getX(), e.getY());
  modelPoint.setLocation(getModelX(e.getX()),
  getModelY(e.getY()));
}

private void setPointBounds(int index1,
 int index2) {
  int boundx1 = inset;
  int boundy1 = componentHeight - inset;
  int boundx2 = componentWidth - inset;
```

```
    int boundy2 = inset;

    if (index1 > -1) {
      boundx1 = getComponentX( ( (StPkt)
       points.get(index1)).getT());
      boundy1 = getComponentY( ( (StPkt)
      points.get(index1)).getD());
    }
    if (index2 < points.size()) {
      boundx2 = getComponentX( ( (StPkt)
       points.get(index2)).getT());
      boundy2 = getComponentY( ( (StPkt)
       points.get(index2)).getD());
    }

    boundRect.setBounds(boundx1, boundy2,
     boundx2 - boundx1, boundy1 - boundy2);
  }

  private int getComponentX(double x) {
    return (int) ( ( (x / gridRangeX) *
    (componentWidth - 2 * inset)) + inset);
  }

  private int getComponentY(double y) {
    return componentHeight -
        (int) ( ( (y / gridRangeY) *
        (componentHeight - 2 * inset)) + inset);
  }

  private double getModelX(int x) {
    return round( ( (double) (x - inset)) /
                 ( (double) (componentWidth
                 - 2 * inset)) * gridRangeX,
                 accuracy);
  }

  private double getModelY(int y) {
    return round( - ( ( (double) (y -
    componentHeight + inset)) /
                     ( (double) (componentHeight
                     - 2 * inset)) * gridRangeY),
                 accuracy);
  }
```

```
  private double round(double d, int ac) {
    return Math.rint(d * Math.pow(10.0, (double) ac)) /
        Math.pow(10.0, (double) ac);
  }

  public void contentsChanged(ListDataEvent e) {
    repaint();
  }

  public void intervalAdded(ListDataEvent e) {
    repaint();
  }

  public void intervalRemoved(ListDataEvent e) {
    repaint();
  }

}
```

9.4.1 SortenEditorPanel.java

```
package gui;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import parser.*;
import java.awt.event.*;

public class SortenEditorFrame extends JFrame
implements ActionListener{

  Kb kb;
  Sorte s;
  int index;
  DefaultListModel iModel = new DefaultListModel();
  DefaultListModel vModel = new DefaultListModel();
  DefaultListModel cModel = new DefaultListModel();
  DefaultListModel fModel = new DefaultListModel();

  BorderLayout borderLayout1   = new BorderLayout();
```

```
JPanel jPanel1               = new JPanel();
JPanel jPanel2               = new JPanel();
JPanel jPanel3               = new JPanel();

JButton jButton1             = new JButton();
JButton jButton2             = new JButton();
JButton jButton3             = new JButton();

FlowLayout flowLayout1       = new FlowLayout();
GridBagLayout gridBagLayout1 = new GridBagLayout();

JButton jButton4             = new JButton();
JRadioButton jRadioButton1   = new JRadioButton();
JRadioButton jRadioButton2   = new JRadioButton();
JButton jButton5             = new JButton();
JScrollPane jScrollPane1     = new JScrollPane();
JList jList1                 = new JList();
JLabel jLabel1               = new JLabel();
JTextField jTextField1       = new JTextField();
JLabel jLabel2               = new JLabel();
JTextField jTextField2       = new JTextField();
JPanel jPanel4               = new JPanel();
FlowLayout flowLayout2       = new FlowLayout();
JTextField jTextField3       = new JTextField();
JLabel jLabel3               = new JLabel();
JTextField jTextField4       = new JTextField();
JLabel jLabel4               = new JLabel();
JPanel jPanel7               = new JPanel();
GridLayout gridLayout1       = new GridLayout();
JButton jButton6             = new JButton();
JTextField jTextField5       = new JTextField();
JButton jButton7             = new JButton();
JScrollPane jScrollPane2     = new JScrollPane();
JList jList2                 = new JList();
GridBagLayout gridBagLayout2 = new GridBagLayout();
JButton jButton8             = new JButton();
JButton jButton9             = new JButton();
GridBagLayout gridBagLayout3 = new GridBagLayout();
JPanel jPanel8               = new JPanel();
JList jList3                 = new JList();
JTextField jTextField6       = new JTextField();
JScrollPane jScrollPane3     = new JScrollPane();
JButton jButton10            = new JButton();
JButton jButton11            = new JButton();
```

```
GridBagLayout gridBagLayout4 = new GridBagLayout();
JPanel jPanel9               = new JPanel();
JList jList4                 = new JList();
JTextField jTextField7       = new JTextField();
JScrollPane jScrollPane4     = new JScrollPane();

TitledBorder titledBorder1;
Border border1;
TitledBorder titledBorder2;
Border border2;
TitledBorder titledBorder3;
Border border3;
ButtonGroup buttonGroup1     = new ButtonGroup();



public SortenEditorFrame(Sorte sort, Kb kb, int
 selectedIndex) throws HeadlessException {

  this.s = sort;
  this.kb = kb;
  this.index = selectedIndex;

  try {
    jbInit();
  }
  catch(Exception e) {
    e.printStackTrace();
  }

  if (s.getKonst() != null) {
    for (int i = 0; i < s.getElements().length;
     i++) {
      iModel.addElement(s.getKonst()[i].getName());
    }

    for (int i = 0; i < kb.getVariablen().size();
    i++) {
      Object o = kb.getVariablen().get(i);

      if (o instanceof parser.Variable) {
        if ( ( (Variable) o).getSorte().getName()
        .equals(s.getName()))
          vModel.addElement(((Variable) o).getName());
```

```
        }
        else if (o instanceof parser.Konstante) {
          if ( ( (Konstante) o).getSorte().getName()
          .equals(s.getName()))
            cModel.addElement(((Konstante) o).getName());
        }
        else if (o instanceof parser.Funktion) {
          if ( ( (Funktion) o).getSorte().getName()
          .equals(s.getName()))
            fModel.addElement(((Funktion) o).
            toString());
        }
      }
    }

    if (s != null)
      jTextField1.setText(s.getName());
    else
      jTextField1.setText("");

    jList1.setModel(iModel);
    jList2.setModel(vModel);
    jList4.setModel(cModel);
    jList3.setModel(fModel);

  }
  private void jbInit() throws Exception {
    titledBorder1 = new TitledBorder(BorderFactory.
    createEtchedBorder(new Color(192, 255, 131),new
    Color(94, 156, 64)),"Variablen");
    border1 = BorderFactory.createCompoundBorder
    (titledBorder1,
    BorderFactory.createEmptyBorder(5,5,5,5));
    titledBorder2 = new TitledBorder(BorderFactory.
    createEtchedBorder(new Color(192, 255, 131),new
     Color(94, 156, 64)),"Konstanten");
    border2 = BorderFactory.createCompoundBorder
    (titledBorder2,
    BorderFactory.createEmptyBorder(5,5,5,5));
    titledBorder3 = new TitledBorder(BorderFactory.
    createEtchedBorder(new Color(192, 255, 131),
    new Color(94, 156, 64)),"Funktionen");
    border3 = BorderFactory.createCompoundBorder
    (titledBorder3,
```

```
BorderFactory.createEmptyBorder(5,5,5,5));
this.getContentPane().setLayout(borderLayout1);
jButton1.setText("Abbrechen");
jButton1.addActionListener(this);
jButton1.setActionCommand("cnl");

jButton2.setText("Fertig");
jButton2.addActionListener(this);
jButton2.setActionCommand("oka");
if (index < 0)
  jButton2.setEnabled(false);

jButton3.setText("Als neue Sorte hinzufügen");
jButton3.addActionListener(this);
jButton3.setActionCommand("new");

jPanel3.setLayout(flowLayout1);
flowLayout1.setAlignment(FlowLayout.RIGHT);
flowLayout1.setHgap(5);
jPanel1.setLayout(gridBagLayout1);
jButton4.setText("Hinzufügen");


jButton4.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
    if (buttonGroup1.getSelection().getActionCommand().
    equals("s")) {
      iModel.addElement(jTextField2.getText());
    } else {
      int i = Integer.parseInt(jTextField4.getText());
      int j = Integer.parseInt(jTextField3.getText());
      for (int k = i; k <= j; k++) {
        iModel.addElement(Integer.toString(k));
      }
    }
  }
});

jButton5.setText("Löschen");
jButton5.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
    if (jList1.getSelectedIndex() > -1) iModel.
    remove(jList1.getSelectedIndex());
  }
```

```
});

jLabel1.setText("Individuen");
jLabel2.setFont(new java.awt.Font("SansSerif", 1,
11));
jLabel2.setPreferredSize(new Dimension(30, 15));
jLabel2.setDisplayedMnemonic('0');
jLabel2.setText("Sorte");
jRadioButton1.setText("");
jRadioButton1.setActionCommand("s");
jRadioButton2.setText("");
jRadioButton2.setActionCommand("z");
jPanel4.setLayout(flowLayout2);
jLabel3.setText("bis");
jLabel4.setText("Zahlen von");
jList1.setMaximumSize(new Dimension(2000, 2000));
jList1.setMinimumSize(new Dimension(60, 100));
jList1.setPreferredSize(new Dimension(100, 300));
jPanel2.setLayout(gridLayout1);
jButton6.setText("Hinzufügen");

jButton6.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
    vModel.addElement(jTextField5.getText());
  }
});

jButton7.setText("Löschen");
jButton7.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
    if (jList2.getSelectedIndex() > -1) vModel.
    remove(jList2.getSelectedIndex());
  }
});

jPanel7.setLayout(gridBagLayout2);
jList2.setMaximumSize(new Dimension(2000, 2000));
jList2.setMinimumSize(new Dimension(50, 100));
jList2.setPreferredSize(new Dimension(100, 200));
jButton8.setText("Hinzufügen");
jButton8.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
    fModel.addElement(jTextField6.getText());
  }
```

```
});

jButton9.setText("Löschen");
jButton9.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
    if (jList3.getSelectedIndex() > -1) fModel.
    remove(jList3.getSelectedIndex());
  }
});

jPanel8.setLayout(gridBagLayout3);
jList3.setMaximumSize(new Dimension(2000, 2000));
jList3.setMinimumSize(new Dimension(50, 100));
jList3.setPreferredSize(new Dimension(100, 200));
jButton10.setText("Hinzufügen");
jButton10.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
    cModel.addElement(jTextField7.getText());
  }
});

jButton11.setText("Löschen");
jButton11.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
    if (jList4.getSelectedIndex() > -1) cModel.
    remove(jList4.getSelectedIndex());
  }
});

jPanel9.setLayout(gridBagLayout4);
jList4.setMaximumSize(new Dimension(2000, 2000));
jList4.setMinimumSize(new Dimension(50, 100));
jList4.setPreferredSize(new Dimension(100, 200));
jScrollPane1.setOpaque(true);
jScrollPane1.setPreferredSize(new Dimension(100, 200));
jPanel7.setBorder(border1);
jPanel9.setBorder(border2);
jPanel8.setBorder(border3);
jTextField2.setText("");
jTextField4.setMinimumSize(new Dimension(20, 21));
jTextField4.setPreferredSize(new Dimension(20, 21));
jTextField4.setText("");
jTextField3.setMinimumSize(new Dimension(20, 21));
jTextField3.setPreferredSize(new Dimension(20, 21));
```

```
jTextField3.setText("");
jTextField5.setText("");
jTextField7.setText("");
jTextField6.setText("");
this.getContentPane().add(jPanel1,  BorderLayout.WEST);
this.getContentPane().add(jPanel2,  BorderLayout.CENTER);
this.getContentPane().add(jPanel3,  BorderLayout.SOUTH);
jPanel3.add(jButton3, null);
jPanel3.add(jButton2, null);
jPanel3.add(jButton1, null);
jPanel1.add(jButton4,
new GridBagConstraints(0, 7, 2, 1, 0.0, 0.0
        ,GridBagConstraints.CENTER,
        GridBagConstraints.NONE, new Insets(0,
         5, 5, 5), 0, 0));
jPanel1.add(jRadioButton1,
 new GridBagConstraints(0, 5, 1, 1, 0.0, 0.0
        ,GridBagConstraints.EAST,
        GridBagConstraints.NONE, new Insets(0,
         0, 0, 0), 0, 0));
jPanel1.add(jRadioButton2,
new GridBagConstraints(0, 6, 1, 1, 0.0, 0.0
        ,GridBagConstraints.EAST,
         GridBagConstraints.NONE, new Insets(0,
          0, 0, 0), 0, 0));
jPanel1.add(jButton5,
 new GridBagConstraints(0, 3, 2, 1, 0.5, 0.0
        ,GridBagConstraints.WEST,

        GridBagConstraints.NONE, new Insets(0,
         5, 5, 5), 0, 0));
jPanel1.add(jScrollPane1,
 new GridBagConstraints(0, 2, 2, 1, 1.0, 1.0
        ,GridBagConstraints.CENTER,
        GridBagConstraints.BOTH, new Insets(0,
         5, 0, 5), 0, 0));
jPanel1.add(jLabel1,        new GridBagConstraints(0
, 1, 2, 1, 0.0, 0.0
        ,GridBagConstraints.CENTER,
         GridBagConstraints.NONE, new Insets(5,
          5, 0, 5), 0, 0));
jPanel1.add(jTextField1,
new GridBagConstraints(1, 0, 1, 1, 0.6, 0.0
        ,GridBagConstraints.CENTER,
```

```
          GridBagConstraints.HORIZONTAL, new Insets(5,
           0, 5, 5), 0, 0));
    jPanel1.add(jLabel2,     new GridBagConstraints(0,
    0, 1, 1, 0.0, 0.0
            ,GridBagConstraints.CENTER,
             GridBagConstraints.NONE, new Insets(5,
              5, 5, 0), 0, 0));
    jPanel1.add(jTextField2,   new GridBagConstraints(1,
     5, 1, 1, 0.0, 0.0
            ,GridBagConstraints.CENTER,
            GridBagConstraints.HORIZONTAL, new Insets(0,
            0, 0, 0), 0, 0));
    jPanel1.add(jPanel4,   new GridBagConstraints(1,
     6, 1, 1, 0.0, 0.0
            ,GridBagConstraints.WEST,
            GridBagConstraints.HORIZONTAL, new Insets(0,
             0, 0, 0), 0, 0));
    jPanel4.add(jLabel4, null);
    jPanel4.add(jTextField4, null);
    jPanel4.add(jLabel3, null);
    jPanel4.add(jTextField3, null);
    jScrollPane1.getViewport().add(jList1, null);
    jPanel2.add(jPanel7, null);
    jPanel7.add(jScrollPane2,   new GridBagConstraints(0,
     0, 1, 1, 1.0, 1.0
            ,GridBagConstraints.CENTER, GridBagConstraints.
            BOTH, new Insets(5, 5, 0, 5), 0, 0));
    jScrollPane2.getViewport().add(jList2, null);
    jPanel7.add(jButton7,   new GridBagConstraints(0,
     1, 1, 1, 0.0, 0.0
            ,GridBagConstraints.WEST,
            GridBagConstraints.NONE, new Insets(0,
             5, 5, 5), 0, 0));
    jPanel7.add(jTextField5,   new GridBagConstraints(0
    , 2, 1, 1, 1.0, 0.0
            ,GridBagConstraints.WEST,
            GridBagConstraints.HORIZONTAL, new Insets(0,
            5, 0, 5), 0, 0));
    jPanel7.add(jButton6,   new GridBagConstraints(0,
    3, 1, 1, 1.0, 0.0
            ,GridBagConstraints.CENTER,
            GridBagConstraints.HORIZONTAL, new Insets(0,
             5, 5, 5), 0, 0));
    jPanel2.add(jPanel9, null);
```

```
jPanel9.add(jScrollPane4, new GridBagConstraints(0,
 0, 1, 1, 1.0, 1.0
        ,GridBagConstraints.CENTER,
        GridBagConstraints.BOTH, new Insets(5,
         5, 0, 5), 0, 0));
jPanel9.add(jButton11, new GridBagConstraints(0,
 1, 1, 1, 0.0, 0.0
        ,GridBagConstraints.WEST,
        GridBagConstraints.NONE, new Insets(0,
         5, 5, 5), 0, 0));
jPanel9.add(jTextField7, new GridBagConstraints(0,
 2, 1, 1, 1.0, 0.0
        ,GridBagConstraints.WEST,
         GridBagConstraints.HORIZONTAL, new Insets(0
          5, 0, 5), 0, 0));
jPanel9.add(jButton10, new GridBagConstraints(0,
 3, 1, 1, 1.0, 0.0
        ,GridBagConstraints.CENTER,
         GridBagConstraints.HORIZONTAL, new Insets(0,
          5, 5, 5), 0, 0));
jScrollPane4.getViewport().add(jList4, null);
jPanel2.add(jPanel8, null);
jPanel8.add(jScrollPane3, new GridBagConstraints(0,
0, 1, 1, 1.0, 1.0
        ,GridBagConstraints.CENTER,
        GridBagConstraints.BOTH, new Insets(5,
         5, 0, 5), 0, 0));
jPanel8.add(jButton9, new GridBagConstraints(0,
 1, 1, 1, 0.0, 0.0
        ,GridBagConstraints.WEST,
        GridBagConstraints.NONE, new Insets(0,
         5, 5, 5), 0, 0));
jPanel8.add(jTextField6, new GridBagConstraints(0,
 2, 1, 1, 1.0, 0.0
        ,GridBagConstraints.WEST,
        GridBagConstraints.HORIZONTAL, new Insets(0,
         5, 0, 5), 0, 0));
jPanel8.add(jButton8, new GridBagConstraints(0,
3, 1, 1, 1.0, 0.0
        ,GridBagConstraints.CENTER,
        GridBagConstraints.HORIZONTAL, new Insets(0,
         5, 5, 5), 0, 0));
jScrollPane3.getViewport().add(jList3, null);
buttonGroup1.add(jRadioButton1);
```

```
    buttonGroup1.add(jRadioButton2);

  }

  public void actionPerformed(ActionEvent e){

    if (e.getActionCommand().equals("cnl")){
      this.dispose();
    } else if (e.getActionCommand().equals("oka")){

      s.setName(jTextField1.getText());

      String[] sa = new String[iModel.size()];
      for (int p=0; p < iModel.size(); p++) sa[p]=
      ((String)iModel.get(p));

      s.setIndividuenBereich(sa);

      for (int i = 0; i < vModel.size(); i++) {
        for (int k = 0; k < kb.getVariablen().size();
         k++){
          Object o = kb.getVariablen().get(k);
          if (o instanceof parser.Variable) {
            if ( ( (Variable) o).getName().equals(
            ((String)vModel.get(i))))
              kb.getVariablen().remove(k);
          }
        }
        kb.getVariablen().add(new Variable((
        (String)vModel.get(i)), s));
      }

      for (int i = 0; i < cModel.size(); i++) {
        for (int k = 0; k < kb.getVariablen().size()
        ; k++){
          Object o = kb.getVariablen().get(k);
          if (o instanceof parser.Konstante) {
            if ( ( (Konstante) o).getName().equals(
            ((String)cModel.get(i))))
              kb.getVariablen().remove(k);
          }
        }
        kb.getVariablen().add(new Konstante((
        (String)cModel.get(i)), s));
```

```
}

for (int i = 0; i < fModel.size(); i++) {
  for (int k = 0; k < kb.getVariablen().size();
   k++){
    Object o = kb.getVariablen().get(k);
    if (o instanceof parser.Funktion) {
      if ( ( (Funktion) o).toString().
      equals(((String)fModel.get(i))))
        kb.getVariablen().remove(k);
    }
  }


  String[] split = ((String) fModel.get(i)).
  split(" ");
  String name = split[0];
  Term[] term = new Term[split.length - 1];

  for (int x = 0; x < term.length; x++) {

    for (int v = 0; v < kb.getVariablen().
    size(); v++) {
      String stringname = kb.getVariablen()
      .elementAt(v).toString();

      if (stringname.equals(split[x + 1])) {

        if ( (kb.getVariablen().elementAt(v)
        instanceof Variable) ||
            (kb.getVariablen().elementAt(v)
            instanceof Konstante) ||
            (kb.getVariablen().elementAt(v)
            instanceof Funktion)) {

          term[x] = (Term) kb.getVariablen().
          elementAt(v);
        }
        else {
          System.out.println(
              "Fehler: Funktion benutzt bisher
               nicht definierte Parameter");
        }
```

```
          }

        }
      }

      kb.getVariablen().add(new Funktion(name ,
       term , s));

    }

    this.dispose();

  } else if (e.getActionCommand().equals("new")){


    String[] sa = new String[iModel.size()];
    for (int p=0; p < iModel.size(); p++) sa[p]=
    ((String)iModel.get(p));


    Sorte neu = new Sorte(jTextField1.getText(), sa);

    kb.getSorten().add(neu);

    for (int i = 0; i < vModel.size(); i++) {
      for (int k = 0; k < kb.getVariablen().size();
      k++){
        Object o = kb.getVariablen().get(k);
        if (o instanceof parser.Variable) {
          if ( ( (Variable) o).getName().equals
          (((String)vModel.get(i))))
            kb.getVariablen().remove(k);
        }
      }
      kb.getVariablen().add(new Variable(((String)
      vModel.get(i)), neu));
    }

    for (int i = 0; i < cModel.size(); i++) {
      for (int k = 0; k < kb.getVariablen().size();
      k++){
        Object o = kb.getVariablen().get(k);
        if (o instanceof parser.Konstante) {
          if ( ( (Konstante) o).getName().equals(
```

```
        ((String)cModel.get(i))))
          kb.getVariablen().remove(k);
      }
    }
    kb.getVariablen().add(new Konstante(((String)
    cModel.get(i)), neu));

  }

  for (int i = 0; i < fModel.size(); i++) {
    for (int k = 0; k < kb.getVariablen().size();
     k++){
      Object o = kb.getVariablen().get(k);
      if (o instanceof parser.Funktion) {
        if ( ( (Funktion) o).toString().
        equals(((String)fModel.get(i))))
          kb.getVariablen().remove(k);
      }
    }


    String[] split = ((String) fModel.get(i))
    .split(" ");
    String name = split[0];
    Term[] term = new Term[split.length - 1];

    for (int x = 0; x < term.length; x++) {

      for (int v = 0; v < kb.getVariablen().
      size(); v++) {
        String stringname = kb.getVariablen()
        .elementAt(v).toString();

        if (stringname.equals(split[x + 1])) {

          if ( (kb.getVariablen()
          .elementAt(v)instanceof Variable) ||
              (kb.getVariablen().
              elementAt(v)instanceof Konstante) ||
              (kb.getVariablen().
              elementAt(v)instanceof Funktion)) {

            term[x] = (Term) kb.getVariablen().
            elementAt(v);
```

```
              }
              else {
                System.out.println(
                    "Fehler: Funktion benutzt bisher
                    nicht definierte Parameter");
              }
            }

          }
        }

        kb.getVariablen().add(new Funktion(name ,
         term , neu));

      }

      iModel.removeAllElements();
      vModel.removeAllElements();
      cModel.removeAllElements();
      fModel.removeAllElements();

      jTextField1.setText("");
      jButton2.setEnabled(false);

      s = new Sorte("neu");

    }
  }


}
```