Simulation von Prüfungsordnungen und Studiengängen mit Hilfe von Constraint-logischer Programmierung

Dereje Tewolde Habtemariam

29. März 2006

eingereicht bei Prof. Dr. Manfred Schmidt-Schauß

Proffesur für Künstliche Intelligenz / Softwaretechnologie

Erklärung gemäß DPO §11 Abs. 11

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, den 29. März 2006

Dereje Tewolde Habtemariam

Danksagung

Mein besonderer Dank geht an Frau Hildegard Rauschmayr, die mir während meiner gesamten Studienzeit zur Seite stand.

Mein Dank geht auch an Prof. Schmidt-Schauß, der es mir ermöglichte diese Arbeit zu schreiben und an meinen Betreuer David Sabel, der für Fragen immer zur Verfügung stand.

Dereje Tewolde Habtemariam

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ziel der Diplomarbeit ist die Simulation der Prüfungsordnung für den neuen Bachelorstudiengang Informatik der Johann Wolfgang Goethe Universität Frankfurt am Main. Mit Hilfe der Simulation sollen Entscheidungen und Festlegungen, die vor der Einführung neuer Studiengänge oder neuer Prüfungsordnungen zu treffen sind, auf praktische Tauglichkeit überprüft werden.
Für die Simulation der Prüfungsordnung wird die Constraint-logische Programmiersprache, ECL^{i}PS^e benutzt. Constraint-logische Programmiersprachen *(Constraint Logic Progamming languages, CLP)* sind [ThSl97] eine Familie von Programmiersprachen, die Ende der achtziger Jahre als natürliche Fusion zweier deklarativen Paradigmen entstand, aus der Logik und der Constraint Programmierung. Wir werden in dieser Arbeit zunächst die logische Programmierung und anschließend die Constraint Programmierung erläutern, bevor wir mit der Simulation der Prüfungsordnung mit ECL^{i}PS^e beginnen.
Logische Programmierung wird oft auch als wissensbasierte Programmierung bezeichnet [Ram95]. In der Sprache der Logik besteht das Wissen aus einer Menge von wahren Aussagen. Gibt der Benutzer eine Behauptung in ein Logik Programm, so versucht das System sie auf der Grundlage des Wissens als richtig oder falsch zu beweisen. Es wird versucht auf der Basis von Wissen über einen speziellen Problembereich Schlussfolgerungen zu ziehen.
Prädikatenlogik wird in der logischen Programmierung zur Beschreibung des zu lösenden Problems und dessen maschinelle Bearbeitung angewendet. Ein wichtiger Vertreter der logischen Programmiersprachen ist Prolog. Prolog gilt als die am meisten eingesetzte und bekannteste logische Programmiersprache im Teilgebiet der Informatik, künstliche Intelligenz.
Gegenstand der Constraint Programmierung ist es, mit Hilfe formaler Constraints reale Objekte und deren Wechselbeziehungen für ein vorgegebenes Problem so zu modellieren, dass das dabei entstehende, formale Modell direkt als Programm zur Lösung des Problems eingesetzt werden kann. Constraint Programmierung wurde zu Beginn in der Bilderkennung eingesetzt [Barták99], wobei Objekte auf der Basis von Linienzeichnungen erkannt werden sollten.

Mit einer Verbindung von Constraint-Techniken und der, in logischen Programmiersprachen integrierten Suchmethoden werden in der Wirtschaft und Technik verschiedene Probleme effizient und deklarativ modelliert und behandelt. ECL^{i}PS^e entstand mit der Einbettung von Constraint-Techniken in die Logische Programmiersprache Prolog. Dabei werden Methoden der Constraint Programmierung verwendet, um den Lösungsraum einzuschränken, bevor man durch einen Suchschritt Alternativen erprobt. Viele der heute verwendeten Systeme der Constraint Programmierung sind Erweiterungen von logischen Programmiersprachen z.B. CHIP, ECL^{i}PS^e, Prolog III. Auch in anderen Programmiersprachen u. a. objektorientierte und funktionale Programmiersprachen wurden Constraints erfolgreich eingebettet.

1.1 Übersicht

In diesem Abschnitt erläutern wir kurz, wie die vorliegende Arbeit gegliedert ist. Er soll einen Überblick geben, welche Themen der Informatik wichtig sind, um die Arbeitsweise von ECL^{i}PS^e und somit die Lösung des behandelten Problems zu verstehen.
In Kapitel 2 wird Prädikatenlogik, deren Syntax und Semantik eingeführt. Auf Grundlage dessen wird in Kapitel 3 die Logische Programmierung mit Prolog erläutert. Anschließend befassen wir uns in Kapitel 4 mit Constraint Programmierung. In Kapitel 5 wollen wir einen allgemeinen Ansatz der Constraint-logischen Programmierung vorstellen. Nach einer kurzen Einführung in ECL^{i}PS^e werden wir in Kapitel 7 zur Beschreibung und Simulation der Studienordnung übergehen. Kapitel 8 ist eine Zusammenfassung dieser Arbeit. Im Anhang findet man den Teil der Studienordnung, der für die Simulation relevant ist und das Simulationsprogramm.

Kapitel 2

Prädikatenlogik erste Stufe

Die Prädikatenlogik erster Stufe (PL1) ist heute die am besten untersuchte und in der Anwendung am weitesten verbreitete Logik [EiOh]. Sie dient darüber hinaus als Basis für viele andere Logiken, die in der Künstlichen Intelligenz und bei der Beschreibung von Programmen und Prozessen eine Rolle spielen.
Die Prädikatenlogik erster Stufe ist eine Erweiterung, der heute als die logische Basis der Digitaltechnik geltenden Aussagenlogik, wobei Quantoren, Funktions- und Prädikatsymbole hinzukommen. Aussagen wie,

„Die Sonne scheint"

die in der Aussagenlogik atomar sind, werden in der Prädikatenlogik in Terme (*sonne*) und Prädikate (*scheint*) aufgelöst und dargestellt als: $scheint(sonne)$.
Die Eigenschaften und Relationen von Objekten werden durch Prädikatensymbole und Funktionssymbole beschrieben. Geltungsbereiche der Relationen und Eigenschaften werden durch Quantoren bestimmt. Durch diese Konzepte sind Sachverhalte beschreibbar, die im Rahmen der Aussagenlogik nicht formuliert werden können. In der Aussagenlogik, auch Prädikatenlogik der Stufe Null genannt (PL0), ist es z.B. nicht möglich auszudrücken, dass gewisse Objekte in Beziehung zueinander stehen. Es ist ebenfalls nicht möglich auszudrücken, dass eine Eigenschaft für alle Objekte einer Menge gilt oder dass ein Objekt mit einer bestimmten Eigenschaft existiert.
Wir können in PL1 beispielsweise folgende Aussagen wie folgt formulieren.

Alle Menschen sind sterblich:	$\forall x(mensch(x) \rightarrow sterblich(x))$
Es gibt sterbliche Menschen:	$\exists x(mensch(x) \wedge sterblich(x))$

Prädikatenlogik führt zusätzlich noch Variablen ein, die für noch nicht bekannte Objekte des so genannten Diskursbereichs (Domäne) stehen. Diese Variablen ähneln den aus der Mathematik bekannten. Sie sind nicht zu verwechseln mit Variablen in prozeduralen Programmiersprachen, die für Speicherstellen stehen.

2.1 Syntax der Prädikatenlogik erster Stufe

Die Menge der Elemente einer formalen Sprache wird als Signatur oder auch Alphabet bezeichnet. Die Signatur der Prädikatenlogik erster Stufe (PL1) besteht, nach [Ram95] , aus:

- einer Menge von Variablen: $x, y, z, ...$,
- einer Menge von Konstanten: $a, b, c, ...$,
- einer Menge von Funktionssymbolen: $f, g, h, ...$,
- einer Menge von Prädikatensymbolen (Relationssymbolen): $P, Q, R, ...$,
- einer Menge von Junktoren oder Hilfszeichen.

Junktoren sind:

$\neg$	Negation
$\wedge$	Konjunktion
$\vee$	Disjunktion
$\rightarrow$	Implikation (syntaktische Folgerung)
$\leftrightarrow$	Äquivalenz
$\forall$	All-Quantor
$\exists$	Existenz-Quantor

Jedem Funktions- und jedem Prädikatensymbol ist eindeutig eine Stelligkeit $k > 0$ zugewiesen, welche die Anzahl ihrer Argumente angibt.

2.1.1 Terme

Die Menge der prädikatenlogischen Terme bezeichnen Objekte und können wie folgt induktiv definiert werden [hamb02]:

1. Jede Variable ist ein Term.
2. Jede Konstante ist ein Term.
3. Falls f ein Funktionssymbol der Stelligkeit k ist, und falls $t_1, ..., t_k$ Terme sind, so ist $f(t_1, ..., t_k)$ ein Term.
4. Es gibt keine anderen Terme, als die, die durch endliche Anwendung der Schritte 1 bis 3 erzeugt werden. Konstanten- und Variablensymbole sind Terme.

Terme, die keine Variablensymbole enthalten, bezeichnet man als Grundterme. Terme entsprechen Objekten, über die Aussagen formuliert werden sollen.

2.1.2 Formeln

Atome bestehen aus Zeichenfolgen und sind logisch unzerlegbare Ausdrücke.
Die Menge der prädikatenlogischen Terme ist wie folgt induktiv definiert:

1. Alle Atome sind Formeln.
2. Wenn F und G Formeln sind und x ein Variablensymbol ist, dann sind auch $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$, $\forall F$ und $\exists F$ Formeln.

Um Klammern zu sparen, vereinbart man folgende Bindungspriorität:

$$\neg, \wedge, \vee, \rightarrow, \leftrightarrow$$

wobei $\neg$ am stärksten bindet.

Komplexe Aussagen werden mit Junktoren und Quantoren aus atomaren Formeln zusammengesetzt. In der Prädikatenlogik erster Stufe wird nur über Variablen und Konstanten quantifiziert. In höheren Stufen kann über Funktionen und Prädikate quantifiziert werden.
Eine Variable x heißt gebunden, wenn sie in einem Geltungsbereich eines Quantors vorkommt. Sie heißt frei, wenn sie nicht gebunden ist.

Beispiel:
„Jeder Mensch liebt einen Anderen" könnte in PL1, mit Hilfe von All- bzw. Existenz-Quantoren, wie folgt geschrieben werden:

$$\underbrace{\forall\, x(mensch(x) \rightarrow \exists \underbrace{y(mensch(y) \wedge liebt(x,y)))}_{Geltungsbereich\ von\ \exists}}_{Geltungsbereich\ von\ \forall}$$

2.2 Semantik von PL1

Die Semantik einer formalen Sprache ordnet den syntaktischen Elementen eine Bedeutung zu. Die Semantik der PL1 wird durch die Angabe einer Interpretationsstruktur definiert.

Definition
Eine Struktur ist ein Paar $S = (U, I)$, wobei U (Universum, Grundmenge, Domäne) eine beliebige, nicht leere Menge ist und I eine Interpretation (Abbildung) ist.
Dabei bildet I

- Variablen und Konstanten auf Elemente der Grundmenge U,
- k-stellige Funktionssymbolen auf k-stellige Funktionen über U,

- k-stellige Prädikatensymbole auf Mengen von k-Tupeln über U und
- prädikatenlogische Formeln auf Wahrheitswerte (wahr und falsch) ab.

Wir sagen eine Struktur S ist passend zu einer Formel F, falls I für alle in F vorkommenden Konstanten, Funktions-, Prädikaten- und Aussagensymbole sowie alle freien Variablen definiert ist.

Definition
Sei F eine Formel und S eine zu F passende Struktur. Die Interpretation von Termen, zusammengesetzten Termen und atomaren Formeln wird folgendermaßen ([hamb02]) definiert:

- Für jede Variable x ist: $S(x) = I(x)$,
- Für jede Konstante a ist: $S(a) = I(a)$,
- Für jede Formel P ist: $S(P) = I(P)$,
- Für jede k Terme $t_1, ..., t_k$ und jedes k-stellige Funktionssymbol ist:

  $$S(f(t_1, ..., t_k)) = I(f)(S(t_1), ..., S(t_k))$$

  $I(f)$ ist eine k-stellige Funktion über U. Diese Funktion wird auf die Liste von Objekten angewendet, die bei der Auswertung der Terme entsteht.
- Für jede Liste von k Termen $t_1, ..., t_k$ und jedes k-stellige Prädikatenformel P ist:

$$S(P(t_1, ..., t_k)) = \begin{cases} 1 & falls\ I(t_1), ..., I(t_n) \in I(P) \\ 0 & sonst \end{cases} \tag{2.1}$$

Die Interpretation zusammengesetzter Formeln wird wie folgt bestimmt.

Definition
Für alle quantorenfreien Formeln F und G, zu denen S passt, ist

1. Falls G die Form $\neg H$ hat, dann gilt $S(G) = 1$, genau dann wenn $S(H) = 0$, und $S(G) = 0$ sonst.
2. Falls G die Form $H \wedge J$ hat, dann gilt $S(G) = 1$, genau dann wenn $S(H) = S(J) = 1$, und $S(G) = 0$ sonst.
3. Falls G die Form $H \vee J$ hat, dann gilt $S(G) = 1$, genau dann wenn $S(H) = 1$ oder $S(J) = 1$, und $S(G) = 0$ sonst.
4. Falls G die Form $H \rightarrow J$ hat, dann gilt $S(G) = 1$, genau dann wenn $S(H) = 0$ oder $S(J) = 1$, und $S(G) = 0$ sonst.

5. Falls G die Form $H \leftrightarrow J$ hat, dann gilt $S(G) = 1$, genau dann wenn $S(H) = S(J)$, und $S(G) = 0$ sonst.

Mit der Erläuterung von Interpretation komplexer Formeln mit Quantoren wollen wir alle prädikatenlogische Formeln behandelt haben. Für die Interpretation von Formeln mit Quantoren muss man die freien Variablen in der Formel eliminieren. Dies geschieht mit der Substitution.

Definition
Eine Substitution ist eine Folge von Ersetzungen x/t einer Variablen x durch einen Term t, wobei x in t nicht vorkommt. Dabei bezeichnet:

- $t_{x/t}$ den Term, der entsteht, wenn jedes Vorkommen von x durch den Term t ersetzt wird, $F_{x/t}$ ist die Formel, die aus F in diesem Fall entsteht,
- $S_{x/t} = (U, I_{x/t})$ diejenige Struktur, die x als t interpretiert und ansonsten komplett mit S übereinstimmt.

Eine Substitution kann auch als Folge von Ersetzungen beschrieben werden, wenn sie mehreren Variablen einen Wert aus U zuordnet.

Nun zur Interpretation von Formeln mit Quantoren:

Definition
Für jede Variable x und jede Formel F, so dass S zu $\forall xF$ und $\exists xF$ passt, ist

$$S(\forall xF) = \begin{cases} 1 & falls\ für\ alle\ d \in U gilt : S_{[x/d]}(F) = 1 \\ 0 & sonst \end{cases} \tag{2.2}$$

$$S(\exists xF) = \begin{cases} 1 & falls\ es\ ein\ d \in U gilt : S_{[x/d]}(F) = 1 \\ 0 & sonst \end{cases} \tag{2.3}$$

Es muss erwähnt werden, dass wenn zwei zu einer Formel passende Strukturen (bezüglich des Universums und der Interpretationen aller verfügbaren Symbole und freien Variablen der Formel) übereinstimmen, dann liefern sie auch für die Formel denselben Wert.

Definition
Eine zu einer Formel F passende Interpretationsstruktur $S = (U, I)$ heißt ein Modell für F, falls $I(F) = 1$ bzw. wenn I die Formel wahr macht.

Beispiel [Ram95] :
Die Formel $g(f(x, a), x)$ mit der Interpretation:

- Das Universum besteht aus den natürlichen Zahlen,

- f steht für die Addition,
- g steht für „ $>$" ,
- Die Formel $g(f(x,a),x)$ steht für $x + a > x$.

Für die Belegung der Konstanten a mit 0 existiert kein Modell für die Formel, denn $x + 0 > x$ ist eine falsche Aussage.
Wenn im obigen Beispiel das Universum aus den natürlichen Zahlen ohne die Null bestehen würde, so würde es für jede Belegung von x und a ein Modell für die Formel geben. Die Formel würde in diesem Fall unendlich viele Modelle haben.

Formeln können nach ihrer Eigenschaft klassifiziert werden [Uwe00] .

Definition
Eine Formel heißt

- allgemeingültig (Tautologie), wenn sie von jeder Interpretation erfüllt wird,
- erfüllbar, wenn sie von wenigstens einer Interpretation erfüllt wird,
- unerfüllbar (widersprüchlich), wenn sie von keiner Interpretation erfüllt wird.

Aus dieser Definition folgt der folgende Zusammenhang: Eine Formel ist genau dann allgemeingültig, wenn ihre Negation ($\neg F$) unerfüllbar ist.

2.2.1 Semantische Äquivalenz

In den nächsten Abschnitten spielen Äquivalenzen eine bedeutende Rolle. Es stellt sich die Frage, wann zwei Formeln äquivalent sind.

Definition Zwei Formeln G und F sind (logisch) äquivalent, falls für jede Struktur S, die zu F und G passt, gilt:

$$S(F) = S(G).$$

Es existieren folgende wichtige semantische Äquivalenzen für PL1:

$(P \wedge P) \equiv P$	(Idempotenz)
$(P \vee P) \equiv P$	(Idempotenz)
$(P \wedge Q) \equiv (Q \wedge P)$	(Kommutativität von $\wedge$)
$(P \vee Q) \equiv (Q \vee P)$	(Kommutativität von $\vee$)
$(P \vee (Q \vee R)) \equiv ((P \vee Q) \vee R) \equiv (P \vee Q \vee R)$	(Assoziativität von $\vee$)
$(P \wedge (Q \wedge R)) \equiv ((P \wedge Q) \wedge R) \equiv (P \wedge Q \wedge R)$	(Assoziativität von $\wedge$)
$(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$	(Erste Distributivgesetz)
$(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$	(Zweite Distributivgesetz)
$\neg(\neg(P) \equiv P$	(doppelte Negation)
$(\neg(P \vee Q)) \equiv (\neg P) \wedge (\neg Q)$	(de Morgan Regel)

$(\neg(P \wedge Q)) \equiv (\neg P) \vee (\neg Q)$ (de Morgan Regel)
$(P \rightarrow Q) \equiv (\neg P) \vee Q$ (Ersetzung von Implikation)
$\neg\forall x P \equiv \exists x \neg P$
$\neg\exists x P \equiv \forall x \neg P$
$(\forall x P \wedge \forall x Q) \equiv \forall (P \wedge Q)$
$(\exists x P \vee \exists x Q) \equiv \exists (P \vee Q)$

2.2.2 Klauselnormalform

Es wäre für die Praxis wünschenswert, wenn alles, was wir in Prädikatenlogik sagen können, nur auf eine Art und Weise zu formulieren ist. Deshalb werden Einschränkungen auf Aussagen gemacht, so dass nur bestimmte Aussageformen erlaubt sind.

Eine Menge von Formeln der Prädikatenlogik kann in eine semantisch äquivalente Menge von Klauseln transformiert werden, indem oben angegebene Äquivalenzen verwendet werden. Dabei sind Klauseln implizit UND- verknüpft. Klauseln sind Mengen von Literale, die implizit ODER- verknüpft sind. Literale sind atomare Formeln oder negierte atomare Formeln. Alle Variablen in Klauseln sind implizit all-quantifiziert.
Wir wollen im Folgenden die Umwandelung einer Prädikatenlogischen Formel in Klauselnormalform anhand eines Beispiels erläutern.
Ausgangsformel:

$$\forall x((\forall y P(x,y)) \rightarrow \neg(\forall y(Q(x,y) \rightarrow R(x,y))))$$

Schritt 1 Äquivalenzen und Implikationen entfernen: Wir eliminieren $\rightarrow$ und erhalten

$$\forall x(\neg(\forall y P(x,y)) \vee \neg(\forall y(\neg Q(x,y) \vee R(x,y))))$$

Schritt 2 Negationen, soweit es möglich ist, in die Formel ziehen: Wir verteilen die Negationen, so dass jede Negation nur auf ein Atom wirkt, und erhalten

$$\forall x(\exists y \neg P(x,y) \vee (\exists y Q(x,y) \vee \neg R(x,y)))$$

Schritt 3 Quantoren ganz nach vorn („pränex- Normalform") bringen: Wir benennen die Variablen um, sodass jede Variable nur einmal quantifiziert wird, und erhalten

$$\forall x(\exists y \neg P(x,y) \vee (\exists z Q(x,z) \wedge \neg R(x,z)))$$

Schritt 4 Existenz- Quantoren beseitigen (Skolemisierung): Wir eliminieren alle existentiellen Quantoren. Wenn ein existentieller Quantor nicht im Geltungsbereich eines universellen Quantors vorkommt, ersetzen wir jedes Auftauchen der quantifizierten Variablen durch eine bisher nicht verwendete Konstante.

$$\exists x P(x) \rightarrow P(a)$$

Wenn ein existentieller Quantor im Gültigkeitsbereich universeller Quantoren vorkommt, dann ist es möglich, dass die existentiell quantifizierte Variable von den universell quantifizierten abhängt. Wir ersetzen sie daher durch eine bisher nicht verwendete Funktion der universell quantifizierten Variablen.

$$\forall x(\exists y P(x,y)) \rightarrow \forall x P(x, f(x))$$

Die Formel, die man nach diesen Umformungsschritten erhält, ist nicht mehr äquivalent, aber erfüllbarkeitsäquivalent zur Ausgangsformel. Diese Form wird auch Skolemform genannt.

Ergebnis für das Beispiel:

$$\forall x(\neg P(x, f_1(x)) \vee (Q(x, f_2(x)) \wedge \neg R(x, f_2(x))))$$

Schritt 5 Konjunktive Normalform herstellen: Alle verbleibenden Variablen sind nun universell quantifiziert. Wir können die universellen Quantoren daher auch fortlassen. Alle Variablen werden implizit als universell quantifiziert betrachtet.

$$\neg P(x, f_1(x)) \vee (Q(x, f_2(x)) \wedge \neg R(x, f_2(x)))$$

Schritt 6 All-Quantoren auf Klauseln verteilen: Wir bringen den Satz in konjunktive Normalform d.h. eine Konjunktion von Disjunktionen.

$$(\neg P(x, f_1(x)) \vee Q(x, f_2(x))) \wedge (\neg P(x, f_1(x)) \vee \neg R(x, f_2(x)))$$

Schritt 7 Entfernen von Operatoren durch schreiben als Mengen: Wir eliminieren alle konjunktiven Verknüpfungen und schreiben die Konjunktion als eine Menge von Klauseln.

$$\{\{\neg P(x, f_1(x)), Q(x, f_2(x))\}\{\neg P(x, f_1(x)), \neg R(x, f_2(x))\}\}$$

Schritt 8 Umbenennung von Variablen, so dass keine Variable in mehr als einer Klausel nicht erscheint:

$$\{\{\neg P(x_1, f_1(x_1)), Q(x_1, f_2(x_1))\}\{\neg P(x_2, f_1(x_2)), \neg R(x_2, f_2(x_2))\}\}$$

2.2.3 Semantische Folgerung

Wenn wir eine Ansammelung wahrer prädikatenlogischer Aussagen haben, auch Axiome genannt, dann wollen wir wissen, welche neuen wahren Aussagen sich als Konsequenz aus ihnen ergeben. Um Fragen dieser Art zu beantworten, werden Verfahren angewendet, die wir mit Inferenzverfahren, Deduktionsverfahren oder Herleitungsverfahren bezeichnen. Ziel dieser Verfahren ist die effiziente Konstruktion von Folgerungen aus gegebenen Formeln auf rein syntaktischer Basis.

Definition
Eine Formel G folgt logisch aus einer Formel F (einer Formelnmenge) falls für jede Struktur S, die zu F und G passt, gilt:

Wenn S ein Modell für F ist, dann ist S auch ein Modell für G. Wir schreiben $F \models G$.

Einen wichtigen Zusammenhang zwischen der semantischen und syntaktischen Folgerungsbeziehung stellt der folgende, als Deduktionstheorem bekannte, Satz [MSS04] .

Satz:
Für alle Formeln F und G gilt: $F \models G$ genau dann, wenn $(F \rightarrow G)$ über alle Interpretationsstrukturen allgemeingültig (Tautologie) ist.

Beweis:
Um die Äquivalenz der beiden Aussagen zu zeigen, prüfen wir, ob eine Aussage die andere impliziert („⇒") und umgekehrt.

1. Zu zeigen ist „$F \models G \Rightarrow (F \rightarrow G$ ist allgemeingültig)"

Es gelte $F \models G$.
Sei I eine beliebige Interpretation.
Wenn F in I wahr ist, dann ist auch G wahr, nach Annahme.
Damit ist aber auch die Formel $F \rightarrow G$ in I wahr.
Wenn F in I falsch ist, dann ist die Formel $F \rightarrow G$ in I wahr.
Also gilt $F \rightarrow G$ in allen Interpretationen I.
Damit ist $F \rightarrow G$ allgemeingültig.

2. Zu zeigen ist „$(F \rightarrow G$ ist allgemeingültig$) \Rightarrow F \models G$ "

Es gelte $F \rightarrow G$ ist allgemeingültig.
Sei I eine beliebige Interpretation.
Wenn I die Formel F erfüllt, dann gilt dies auch für G, da $F \rightarrow G$ allgemeingültig ist. Da die Interpretation beliebig gewählt war und für alle Interpretationen gilt, gilt somit auch $F \models G$ ▫

Die Semantische Folgerung wird somit auf die Unerfüllbarkeitsfrage reduziert.

2.2.4 Unentscheidbarkeit von PL1

Wir wollen in diesem Abschnitt die Begriffe Entscheidbarkeit und semi-Entscheidbarkeit erläutern. Dabei wollen wir weder Beweise führen noch ins Detail gehen.

Definition
Unter Entscheidbarkeit von Problemen verstehen wir die Existenz eines deterministischen Verfahrens (Algorithmus), welches auf die Eingabe einer Fragestellung des Problems nach endlich vielen Schritten stets terminiert und die Lösung mit „ja" oder „nein" liefert.

In der Prädikatenlogik gibt es erfüllbare Formeln, die nur unendlich viele Modelle besitzen. Es ist dadurch nicht möglich auf natürlicher Art z.B. mit Hilfe der Wahrheitstafelmethode alle möglichen Strukturen in endlicher Zeit zu betrachten und zu prüfen, ob Formeln erfüllbar sind. Die Frage, ob eine gegebene Formel in PL1 allgemeingültig ist, ist unentscheidbar. Dies wird in [Uwe00] mit Hilfe des so genannten Postschen Korrespondenzproblem bewiesen.

Aus der Tatsache, dass Allgemeingültigkeit einer Formel in PL1 nur mit der Unerfüllbarkeit ihrer Negation möglich ist, kann gezeigt werden, dass auch Erfüllbarkeit nicht entscheidbar sein kann. Denn die Existenz eines Verfahrens für die Erfüllbarkeit würde zur Existenz eines Verfahrens für die Allgemeingültigkeit führen.

Definition
Ein Problem heißt semi-Entscheidbar, wenn man einen Algorithmus angeben kann, der genau auf denjenigen Eingaben nach endlicher Zeit terminiert, für die die zugrunde liegende Fragestellung des Problems mit „ja" zu beantworten ist.

Sowohl Allgemeingültigkeit als auch Unerfüllbarkeit von Formeln in PL1 ist semi-Entscheidbar. Der so genannte Gilmore Algorithmus liefert ein semi-entscheidbares Verfahren für die Unerfüllbarkeit von Formeln in der PL1 in Skolemform [Uwe00] .

2.2.5 Unifikation und Prädikatenlogische Resolution

Wir wollen in diesem Abschnitt ein semi-Entscheidungsverfahren, Resolution, für die Unerfüllbarkeit prädikatenlogischer Formeln vorstellen, das in der vorliegenden Arbeit von Bedeutung ist. Bevor wir Resolution einführen, wollen wir zunächst den Begriff Unifikation erläutern.

Unifikation

Man versteht unter Unifikation zweier Terme eine solche Substitution der in ihnen vorkommenden Variablen durch Terme, dass beide identische Zeichenfolgen sind. Zwei Terme T_1 und T_2 können durch Substitution von Variablen unter folgenden Umständen gleichgemacht werden:

- Sind T_1 und T_2 Konstanten, dann unifizieren sie nur, wenn sie das gleiche Objekt sind.

- Ist T_1 eine Variable, dann kann T_2 beliebig sein, beide unifizieren. T_1 wird zu T_2 substituiert.
- Sind T_1 und T_2 zusammengesetzte Terme, dann erfolgt eine Unifikation nur dann, wenn beide den gleichen Funktor haben und alle einander entsprechenden Argumente ihrerseits unifizieren.

Für Unifikation von Variablen muss gelten:

- sind beide Variablen ungebunden, dann stehen sie für den gleichen Wert, d. h. erhält eine Variable einen Wert, wird dieser automatisch der zweiten Variablen zugewiesen.
- ist eine Variable gebunden und die andere nicht, so erhält die freie Variable den Wert der gebundenen.
- sind beide Variablen gebunden, so ist die Unifikation gleich der von Termen.

Definition
Eine Substitution, die eine Menge von Literalen bis auf ihr Vorzeichen syntaktisch gleich macht, heißt **Unifikator**. Eine Substitution sub heißt allgemeinster Unifikator für eine Menge von Literalen, genau dann wenn

1. *sub* ein Unifikator für die Menge der Literalen ist und
2. zu jedem Unifikator *sub* eine Substitution

   $$sub''$$

   existiert, so dass gilt:

   $$sub = sub''(sub)$$

Anders formuliert sub ist ein allgemeinster Unifikator, genau dann, wenn jeder Unifikator sich durch Spezialisierung aus ihm gewinnen lässt.

Wir möchten an dieser Stelle nur erwähnen, dass jede unifizierbare Literalmenge laut Unifikationssatz von Robinson [KI02] einen allgemeinsten Unifikator hat, so dass man einen Algorithmus angeben kann, der stets terminiert und den allgemeinen Unifikator liefert.

Prädikatenlogische Resolution

Resolution ist eine der wichtigsten Inferenzmethoden, da sie das mechanische Beweisen von Aussagen aus einer vorgegebenen Formelmenge erlaubt und damit für die Praxis interessant ist.

Im den kommenden Kapiteln werden wir uns nur mit einer Variante der Resolution (SLD- Resolution) beschäftigen. Wir werden deshalb eine kurze Einführung in die allgemeine Prädikatenlogische Resolution machen, bevor wir im nächsten Kapitel mit der logischen Programmierung fortfahren.

Bei der Resolution wird zunächst die Negation $\neg F$ der zu beweisenden Formel F angenommen. Es wird anschließend nachgewiesen, dass die Konjunktion aus den vorgegebenen Formelmengen und $\neg F$ unerfüllbar ist. Aufgrund der Annahme, dass die Formelmenge wahr ist, wird die Unwahrheit von $\neg F$ und damit die Wahrheit von F erschlossen.
Grundidee der Resolution ist, dass wir aus den Aussagen

$$P \vee Q$$
$$\neg P \vee R$$

die wahre Aussage

$$Q \vee R$$

schließen. Denn entweder ist P wahr, dann muss R wahr sein, oder P ist falsch, dann muss Q wahr sein. Somit ist in jedem Fall $Q \vee R$ wahr.

Definition
Wenn $(P_1 \cup Q)$ und $(P_2 \cup \neg Q)$ Klauseln sind und wenn in $(P_1 \cup P_2)$ keine Variable negiert und nicht negiert vorkommt, so heißt $(P_1 \cup P_2)$ Resolvente von $(P_1 \cup Q)$ und $(P_2 \cup \neg Q)$.

Es wird mit einem positiven und einem negativen Literal resolviert, wobei der Unifikator auf alle Literale der Resolvente angewandt wird. Die Resolvente enthält alle Literale der beiden Ausgangsklauseln ohne die Literale, mit denen resolviert wurde.

Es wird mit einem positiven und einem negativen Literal resolviert, wobei der Unifikator auf alle Literale der Resolvente angewandt wird. Die Resolvente enthält alle Literale der beiden Ausgangsklauseln ohne die Literale, mit denen resolviert wurde.

Der Beweis für die Unerfüllbarkeit der prädikatenlogischernFormel F aus einer vorgegebenen Formelmenge S mittels Resolution verläuft wie folgt:
Wir transformieren die Formelmenge S und die Negation $(\neg F)$ der zu beweisenden Formel in Klauselnormalform (siehe oben). Wir vereinigen beide Klauselmengen und ordnen Klauseln so an, dass die nicht negierten Literale links und die negierten rechts stehen. Taucht in zwei Klauseln dasselbe Literal in entgegen gesetzten Seiten auf, dann wird resolviert. Treffen zwei Klauseln mit jeweils einem Literal so aufeinander, so entsteht die leere Klausel $(\square)$ die Widerspruchsklausel genannt wird. Das Verfahren terminiert und F ist unerfüllbar. Anderenfalls wird erneut versucht eine neue Resolvente zu finden. Wenn keine neuen Resolventen gebildet werden können, so ist F erfüllbar.

Beispiel:
Gegeben: Eine Formel in KNF

$$(\neg C \vee A) \wedge C \wedge (\neg A \vee \neg B \vee \neg D) \wedge B \wedge (\neg G \vee D) \wedge G \wedge \neg E$$

Die Resolutionsableitung kann als Baum von Klauseln in der Mengendarstellung wie folgt dargestellt werden:

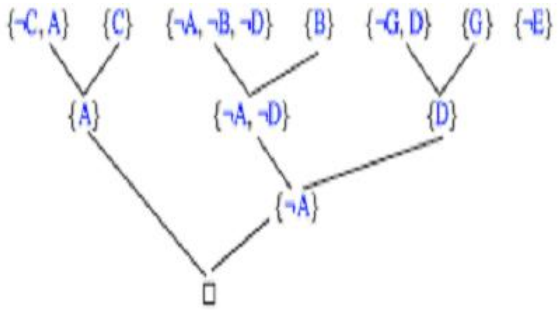

Abbildung 2.1:

Die Begriffe Vollständigkeit und Korrektheit einer Ableitungsregel spielen eine zentrale Rolle.

Definition

- Ein Beweisverfahren heißt vollständig, wenn jede logische Folgerung einer Formel F aus einer Formelmenge mit Hilfe der Ableitungsregel gezeigt werden kann.
- Ein Beweisverfahren heißt korrekt, wenn jede Formel F, die aus einer Formelmenge mit Hilfe des Verfahrens abgeleitet wird, auch eine logische Folgerung ist.

Prädikatenlogische Resolution ist nach [hamb02] korrekt und vollständig. Wir können also immer mittels Resolution die Unerfüllbarkeit einer Formel zeigen, wenn sie tatsächlich unerfüllbar ist. Die Resolution sagt jedoch nicht, welche der verschiedenen Möglichkeiten der Unifikation und der Resolventenbildung als nächstes angewendet werden. Daraus entstehen folgende Situationen:

- Eine Klausel kann während der Resolution mehrmals erzeugt werden,
- es können Klauseln erzeugt werden, die keine spätere Verwendung finden und damit die Anzahl der Klausel erhöhen.

Es gibt zahlreiche Erweiterungen der prädikatenlogischen Resolution, um den Suchraum der kombinierbaren Klauseln einzuschränken wie z. B. Einheitsresolution, Input-Resolution und Subsumtions-Resolution.

Kapitel 3

Logische Programmierung mit Prolog

Programmiersprachen kann man grob einteilen in imperative und deklarative Sprachen. In deklarativen Programmiersprachen gibt es wiederum funktionale und logische Programmiersprachen. Bei den imperativen Sprachen versucht der Programmierer dem Rechner zu beschreiben, „was" das Problem ist und „wie" es gelöst werden soll. In der deklarativen Art dagegen wird lediglich beschrieben, „was" das Problem ist. Diese Methode erlaubt das problemorientierte, und nicht Plattform abhängige, Lösen von Aufgaben, wodurch Lösungsfindung leichter wird.

Prolog basiert auf Prädikatenlogik der ersten Stufe und arbeitet nur mit Horn-Klauseln. Hornklauseln sind Klauseln mit höchstens einem positiven Literal. Die Beschränkung auf Horn-Klauseln bedeutet nur eine geringe Einschränkung der Ausdrucksfähigkeit [Uwe00] und steigert die Effizienz bei der Auswertung.
Das Ausführen eines logischen Pogramms kann als das Herleiten der leeren Klausel aus einer gegebenen Klauselmenge, unter Benutzung von Verfeinerungen der im letzten Kapitel vorgestellten Resolution, angesehen werden.

3.1 Syntax

Es gibt mehrere Prolog-Systeme die sich nach der Art der Implementierung unterscheiden. Wir wollen in diesem Abschnitt eine allgemeine Prolog-Syntax angeben, ohne uns auf ein spezielles Prolog-System zu beziehen. Die hier angegeben Beispiele wurden jedoch mit dem System ECL^{i}PS^e-Prolog getestet. Der Interpreter für ECL^{i}PS^e-Prolog lässt sich über die Umgebung der Constrain-logische Programmiersprache ECL^{i}PS^e aufrufen.

Prolog-Programme setzen sich aus Termen zusammen.
Ein Term kann eine Konstante, eine Variable oder eine Struktur sein. Jeder Term ist eine Folge von Zeichen. Zeichen sind Kleinbuchstaben, Großbuchstaben, Ziffern oder Sonderzeichen. Sonderzeichen sind u.a. (,), ?, ., :, >, <, +.

- **Konstanten**

  Konstanten sind Atome oder Zahlen. Atome sind Zeichenfolgen, die mit Kleinbuchstaben beginnen oder mit Apostrophen eingeschlossen sind. Zahlen in Prolog sind entweder ganze Zahlen oder Fließkommazahlen, z.B. abraham, Bart+, 4

- **Variablen**

  Variablen sind Zeichenfolgen, die mit Großbuchstaben beginnen und bezeichnen nicht-instanzierte (ungebundene) Argumente. Sonderzeichen sind mit Ausnahme von' _ , auch anonyme Variable genannt, nicht in Variablennamen erlaubt. Das erste Auftreten einer Variablen in einer Klausel legt ihren Wert innerhalb dieser Klausel fest. Der Gültigkeitsbereich von Variablen bezieht sich immer nur auf eine Klausel. Außerhalb einer Klausel kann eine Variable gleichen Namens als Platzhalter für ein ganz anderes Objekt stehen.

- **Struktur**

  Eine Struktur ist ein zusammengesetzter Term aus einem Funktor und einer Reihe von Argumenten, die in runde Klammern eingeschlossen und durch Kommata voneinander getrennt sind. Namen von Funktoren beginnen mit Kleinbuchstaben. Argumente können Konstanten, Variablen oder Strukturen sein. Jedem zusammengesetzten Term wird eine Zahl zugeordnet, die Stelligkeit genannt wird und für die Anzahl der Argumente steht. Es ist üblich Prädikate in Form p/i anzugeben, wobei p der Funktor und i die Stelligkeit ist.

Programmieren in Prolog besteht aus dem Deklarieren von Fakten (Tatsachenklauseln), Definieren von Regeln (Programmklauseln) und Stellen von Anfragen (Zielklauseln).

Fakten
Alle wahren Aussagen ohne Vorbedingung werden als Fakten bezeichnet.
Beispiel:
Hier wird eine kleine Wissensbasis erstellt, indem drei Fakten formuliert werden:

$$vater(abraham, isaac).$$
$$männlich(abraham).$$
$$weiblich(sarah).$$

Fakten entsprechen Klauseln mit genau einem positiven Literal, z.B. $\{P\}$.

Regeln
Eine Regel ist eine logische Aussage der Form:

„Wenn C_1 und C_2 und ... und C_n, dann P"

wobei $C_1, C_2, ..., C_n$ die Prämissen (Voraussetzungen) und P die Konklusion (Folgerung) darstellt. In Prolog schreibt man eine solche Regel in der Form:

$$P \text{ :- } C_1, C_2, ..., C_n.$$

Die Darstellung in Klauselnormalform ist:

$$\{P, \neg C_1, \neg C_2, ..., \neg C_n\}.$$

Dabei ist $((C_1 \wedge C_2 \wedge ... \wedge C_n) \rightarrow P)$ die logische Interpretation der Regel.

Beispiel:
$sohn(isaac, abraham)$:- $vater(abraham, isaac), männlich(isaac)$.

Anfragen
Die Notation für eine Anfrage in Prolog ist:

$$?\text{- } C_1, C_2, ..., C_n.$$

Die Darstellung in Klauselnormalform ist:

$$\{C_1, \neg C_2, ..., \neg C_n\}.$$

In Manchen Literaturen wird eine Anfrage auch als ein Ziel bezeichnet. Wir bevorzugen in dieser Arbeit die Bezeichnung Ziel.
Ziele dienen der Ermittlung, ob eine Relation gilt, bzw. ob ein gegebenes Ziel eine logische Konsequenz der Programmklauseln ist. Ist dies der Fall, so antwortet Prolog mit *Yes*, andernfalls mit *No*. Die Antwort *No* bedeutet dabei nicht, dass die Relation nicht gilt, sondern dass sie nicht aus dem Programm abgeleitet werden kann.
Beispiel:
Wir können Ziele auf die Wissensbasis aus dem obigen Beispiel wie folgt stellen.

?- $vater(abraham, isaac)$.
Yes

?- $weiblich(abraham)$.
No

3.2 Semantik

Die Aufgabe eines Prolog-Systems ist zu beweisen, ob ein gegebenes Ziel logisch aus den Programmklauseln ableitbar ist. Die Resolution stellt die Grundlage für eine automatische Beweisführung dar. Prolog nutzt zur Antwortgenerierung ein spezielles Resolutionsverfahren, das unter der Abkürzung SLD-Resolution bekannt ist und einen vollständigen und korrekten Widerlegungskalkül für Horn-Klauselmengen definiert. An dieser Stelle möchte ich auf den Beweis der Vollständigkeit und Korrektheit der SLD-Resolution in [Uwe00] verweisen. Die einzelnen Buchstaben (S, L, D) beschreiben dabei folgende drei Eigenschaften dieser Variante der Resolution.

- S steht für *„Selection rule driven"* und besagt dass, bei jedem Ableitungsschritt nach einer speziellen Auswahlregel eine bestimmte atomare Formel ein Teilziel selektiert wird. Wie genau selektiert wird hier nicht gesagt.
- L steht für *„Linear resolution "* und besagt, dass immer die zuletzt gebildete Resolvente als Eingabe für weitere Resolutionsschritte dient.
- D steht für *„Definite clauses"* und besagt, dass mit einer speziellen Form von Klauseln gearbeitet wird, nämlich mit Horn-Klauseln.

3.2.1 SLD-Bäume

Der Ablauf der Lösungssuche kann durch einen SLD-Baum übersichtlich dargestellt werden. In [Ram95] findet man folgende Definition eines SLD-Baums.

Definition
Sei ein Programm P, und ein Ziel G gegeben.
Ein SLD-Baum für $P \cup \{G\}$ ist ein Baum der folgendes erfüllt:

1. Jeder Knoten ist markiert mit einem Ziel,
2. die Wurzel ist markiert mit G,
3. in jedem Knoten mit nicht leerem Ziel wird ein Atom A des Ziels ausgewählt. Die Söhne dieses Knoten sind dann die möglichen Ziele nach genau einem SLD-Resolutionsschritt mit einer Regel in P,
4. Knoten, die mit der leeren Klausel markiert sind, sind Blätter,
5. in Blatt ist entweder mit dem leeren Ziel markiert, oder es gibt von dem Blatt aus keine SLD-Resolution.

Die Kanten eines SLD-Baums entsprechen Ableitungen. Erfolgreiche Widerlegungen sind Erfolgspfade von der Wurzel bis zu Blättern, die mit einem leeren Ziel markiert sind. Fehlschläge entsprechen Pfaden zu Blättern, die mit einem nicht leeren Ziel markiert sind. Die Menge aller Ableitungen für ein Ziel bildet den dazugehörigen SLD-Baum. Eine Unendliche Ableitung entspricht einem unendlichen Pfad auf dem SLD-Baum.

Bei der SLD Resolution treten zwei Arten von Nichtdeterminismus auf:

1. Es besteht die Möglichkeit verschiedene Literale aus der Zielklausel auszuwählen. Dies ist ein sogenannter *Don't Care*-Nichtdeterminismus, d. h. die Auswahl beeinflusst nicht das Rechenergebnis. Sie beeinflusst lediglich die Länge der Ableitungsschritte.

2. Es besteht die Möglichkeit verschiedene Programmklauseln auszuwählen. Dies ist ein sogenannter *Don't know*-Nichtdeterminismus. Die Auswahl der nächsten Programmklausel ist sehr kritisch und entscheidet darüber, ob eine erfolgreiche Rechnung zustande kommt oder nicht.

SLD-Resolution bleibt bei einer Implementierung korrekt, wenn die Unifikation mit Vorkommenstest (*occur check*) durchführt wird d.h. wenn geprüft wird, ob Variablen mit Ausdrücken unifiziert werden, die dieselben Variablen als Teilausdruck enthalten. Aus Effizienzgründen machen nicht alle Prolog- Interpreter bei der Berechnung eines allgemeinsten Unifikators diesen Test. Das Auslassen dieser Prüfung kann in bestimmten Situationen zur Berechnung fehlerhafter Unifikatoren führen oder dazu, dass der Unifikationsprozess nicht terminiert.

Eine deterministische Auswertungsstrategie muss bei der Implementierung die Nichtdeterminismen auflösen und den gesamten Suchbaum eines Logikprogramms systematisch durchlaufen.

Ein SLD-Baum kann mit den Graphen- Durchsuchungsmethoden Tiefensuche und Breitensuche vollständig durchlaufen werden. Bei der Breitensuche eines Baums werden Knoten des Baums nach ihrer Entfernung von der Wurzel geordnet durchsucht. Zuerst werden alle von der Wurzel direkt durch eine Kante erreichbaren Knoten bearbeitet, danach die mit zwei Kanten Entfernung, dann die mit drei usw.

Der Auswertungsalgorithmus von Prolog arbeitet mittels Tiefensuche in Kombination mit *Backtracking* (Rücksetzen). Unter *Backtracking* versteht man ein Lösungsverfahren, bei dem man versucht eine gefundene Teillösung eines gestellten Problems zu einer Gesamtlösung auszubauen, wobei Entscheidungen rückgängig gemacht werden können, wenn diese nicht zum Erfolg führen.
Bei der Auswertung eines Zieles in Prolog wird der dazu gehörige SLD- Baum beginnend mit der Wurzel, jeweils der am weitesten links stehende Zweig vollständig durchsucht, bevor der nächste Zweig betrachtet wird. Wird keine passende Programmklausel bei der Unifikation gefunden, so springt das Prolog-System durch das *Backtracking* auf den letzten Punkt zurück, an dem eine Entscheidung getroffen wurde. Das System hebt dann die an dieser Stelle durch die Unifikation erfolgten Variablenbindungen auf und wählt die nächste alternative Klausel aus. Können alle Teilziele erfolgreich bearbeitet werden, so gibt Prolog die durchgeführten Variablenbindungen zurück und antwortet mit *Yes*. Schlägt die Bearbeitung eines Zieles fehl und gibt es keine möglichen Alternativen mehr, so antwortet Prolog mit *No*.

Wir wollen im Folgenden ein Beispiel einführen um die Auswertung eines Zieles mittels SLD-Resolution zu erläutern. Zunächst erstellen wir eine kleine Datenbank mit zwei Regeln (1 und 2) und sieben Fakten (3 bis 9), um anschließend ein Ziel zu stellen. Wir haben die Programmklauseln nummeriert, so dass wir später erkennen, welche Klausel angewendet wird. Somit gehören die Zahlen nicht zum Programm.

Beispiel:

$tante(A, B)$:- $weiblich(A), mutter(C, B), schwester(A, C).$ (1)
$tante(A, B)$:- $weiblich(A), vater(C, B), schwester(A, C).$ (2)
$weiblich(tina).$ (3)
$weiblich(gina).$ (4)
$mutter(gina, daniel).$ (5)
$vater(tom, samuel).$ (6)
$schwester(tina, gina).$ (7)

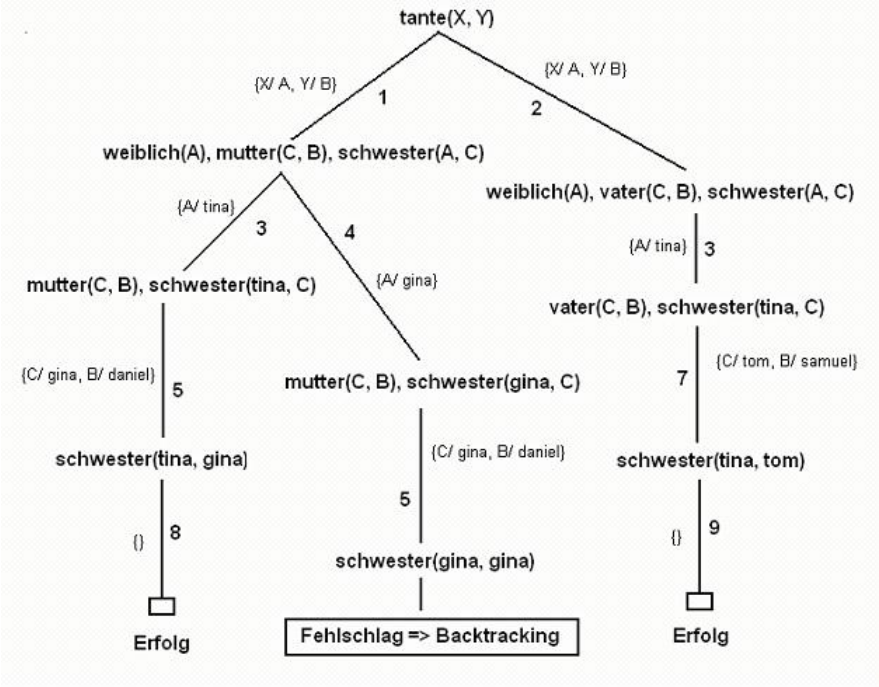

Abbildung 3.1:

Das Ziel „?- $tante(X, Y)$." soll alle möglichen Kombinationen (X, Y) liefern, wobei die Beziehung „X ist Tante von Y" gilt.
Zu diesem Aufruf bieten sich die ersten beiden Klauseln zum Resolvieren an. Wie oben bereits erwähnt, wird das Tiefensuche-Verfahren zur Bestimmung der zu verwendenden Klausel eingesetzt. Somit wird zunächst Klausel (1) ausgewählt, da sie im Programm vor Klausel (2) steht. Wegen dem *Backtracking*-Verfahren wird die Auswahlmöglichkeit gespeichert, um später eventuell darauf zuzugreifen. Zum effizienten Rücksetzen der Variablen wird ein Stack (Keller) verwendet, so dass immer auf die zuletzt gespeicherte Auswahlmöglichkeit zugegriffen wird.

Nach dem Unifizieren mit dem allgemeinsten Unifikator $\{X/A\}$ ergibt sich $weiblich(A)$, $mutter(C, B)$, $schwester(A, C)$ als Resolvente. Nun müssen alle drei Teilziele erfüllt werden. Bei mehreren Teilzielen wird das am weitesten links stehendes Ziel ($weiblich(A)$) ausgewählt. Für das Teilziel $weiblich(A)$ stehen Klauseln 3 und 4 als Resolventen zur Verfügung. Diese Auswahlmöglichkeit wird ebenfalls gespeichert. Da $weiblich(A)$ am weitesten links steht, wird durch den allgemeinsten Unifikator $\{A/tina\}$ mit der Klausel 3 unifiziert. Die nächsten zu erfüllenden Teilziele sind $mutter(C, B)$, $schwester(tina, C)$. Das Teilziel $mutter(C, B)$ wird durch den Unifikator mit Klausel 5 unifiziert. Das letzte zu erfüllende Teilziel $schwester(tina, gina)$ entspricht der Klausel 9. Deshalb antwortet Prolog mit yes und die Variablenbindungen $X = tina$, $Y = daniel$ werden ausgegeben.
Wenn wir weitere Ergebnisse haben möchten, so können wir ein Semikolon eingeben und damit ein *Backtracking* auslösen. Das System springt dann zum letzten Entscheidungspunkt, wo die Variable A mit $tina$ belegt wurde. Nun bekommt A den Wert $gina$ und das System versucht die entstehenden Teilziele zu erfüllen, bis für das Teilziel $schwester(gina, gina)$ keine passende Klausel gefunden wird. Das System wechselt in den *Backtracking*-Modus, um an den letzten Entscheidungspunkt zu gelangen. Dieser Punkt entspricht der Wurzel des SLD-Baums. Nun wird $tante(X, Y)$ mit der Klausel 2 resolviert. Nochmals werden alle drei Teilziele analog zum den linken Teilbaum nacheinander erfüllt. Es entstehen keine Verzweigungen bzw. Auswahlmöglichkeiten mehr und das System liefert die Variablenbindungen $X = tina$, $Y = samuel$. Der Suchraum ist damit vollständig durchsucht.

Das Suchverfahren Tiefensuche ist effizienter als sein Gegenstück Breitensuche, da weniger Speicherplatz benötigt wird und im Durchschnitt eine bessere Laufzeit erzielt werden kann. Tiefensuche kann aber auch in eine endlose Schleife geraten und liefert damit keine Antwort, obwohl Lösungen durch die Resolution ableitbar sind. Dies passiert beispielsweise bei der Benutzung rekursiver Regeln.

Beispiel:

$p(X, Z)$:- $q(X, Z)$.
$p(X, Z)$:- $p(X, Y), q(Y, Z)$.
$q(a, b)$.
$q(b, c)$.

Ziel: ?- $p(a, c)$
Antwort: *Yes*

Die Vertauschung der Reihenfolge der ersten beiden Klauseln führt allerdings zu Endlosschleife:

$p(X, Z)$:- $p(X, Y), q(Y, Z)$.
$p(X, Z)$:- $q(X, Z)$.
$q(a, b)$.

$q(b, c)$.

Ziel: ?- $p(a, c)$

Hier liefert das System keine Lösung, da immer wieder versucht wird p zu erfüllen. Der Keller für das *Backtracking*-Verfahren wird hiermit zu groß und das System muss manuell unterbrochen werden.

Durch die Implementierung geht somit die Vollständigkeit der SLD-Resolution verloren. Das hat zur Folge, dass die Semantik eines Programms durch die deklarative Bedeutung nicht vollständig charakterisiert werden kann. Man unterscheidet daher zwischen der deklarativen und prozeduralen Semantik. Die deklarative Semantik erfasst, welche Ziele aus der gegebenen Klauselmenge sich ableiten lassen lassen, ohne sich auf den Herleitungsvorgang zu beziehen. Die prozedurale Semantik dagegen beschreibt, wie Antworten gefunden werden. Man benötigt zur Analyse und zum Entwurf von Programmen in Prolog genaue Information über die Resolutionsmethode.

3.3 Listen

Die wichtigste Datenstruktur in Prolog ist die Liste. Eine Liste wird syntaktisch als ein Term in eckigen Klammern dargestellt. Die einzelnen Elemente der Liste sind wiederum durch Kommata getrennte Terme (eventuell ebenfalls Listen).
Der Operator | dient dem Aufteilen von Listen in Kopfelement und Rest.
Folgende drei Notationen von Listen sind in Prolog alle gleichwertig:

$.(a, [])$	$[a\|[]]$	$[a]$
$.(a, .(b, []))$	$[a\|[b\|[]]]$	$[a, b]$
$.(a, .(b, .(c, [])))$	$[a\|[b\|[c\|[]]]]$	$[a, b, c]$

Prolog stellt eine Fülle von Systemprädikaten (built-in Prädikate), die vom implementierenden System direkt unterstützt werden. Programmierer ersparen sich das Definieren von Klauseln, um die gewünschte Funktionalität zu erzielen. Zum Bearbeiten von Listen stellt Prolog u.a. folgende Systemprädikate zur Verfügung.

Beispiele:

?- $length([1, 3, a, 8], k)$
$k = 4$
Yes

?- $member(s, [1, 3, s, [a, b, c]])$
Yes

3.4 Arithmetik

Prolog bietet viele eingebaute Prädikate für Arithmetik sowie für Ein- und Ausgabe, deren Code bei Aufruf direkt ausgeführt wird. u.a. folgende arithmetische Prädikate:

$+$, - , * , \ , *mod*, *is*

Im Gegensatz zu anderen Programmiersprachen werden in Prolog arithmetische Ausdrücke nicht sofort ausgeführt, sondern sind gewöhnliche zusammengesetzte Terme.
Beispiel:

?- $X = 2 + 3$.
$X = 2 + 3$

Das Ziel $X = 2 + 3$ bewirkt nur die Bindung von X an den Term $2 + 3$, weil das Funktionssymbol $+$ nicht interpretiert wird. Um arithmetische Ausdrücke explizit zu berechnen, gibt es in Prolog den Infix-Operator *is*, Der *is*-Operator nimmt eine Variable und einen Term als Argumente. Der Term wird berechnet und an die Variable gebunden. Im rechten Argument dürfen keine ungebundenen Variablen vorkommen, da sonst ein Fehler auftritt.
Beispiel:

?- X *is* $2 + 3$.
$X = 5$

?- X *is* $Y + 3$.
Fehler

Prolog bietet folgende Vergleichsprädikate:

$<, >, \leq, \geq, =:=, = \backslash =$

Dabei stehen $=:=$ und $= \backslash =$ für Gleichheit bzw. Ungleichheit. Für die Berechnung der Vergleichsprädikate gilt ebenfalls die Einschränkung, dass kein Argument ungebundene Variable enthält.

Beipiel:

?- $6 * 2 =:= 3 * 4$.
Yes

?- $X > 2$.
Fehler

3.5 Kontrollstrukturen

Prolog bietet die Systemprädikate ! (*cut*-Prädikat), *true* und *fail*, mit denen Programmierer Einfluss auf den Ablauf von Programmen nehmen können. Mit dem *cut*-Prädikat kann *Backtracking* verhindert und somit Effizienzsteigerung erzielt werden. Ein *cut* ist immer wahr und schneidet alle alternativen Lösungen des Ziels aus dem

Suchraum, die in der Klausel links vom *cut* stehen. D.h. die Variablenbelegungen vor dem *cut* können nicht mehr aufgehoben werden. Ein *cut* beeinflusst nicht die Ziele zu seiner Rechten.

Man unterscheidet zwischen grünen und roten *cut*:

- grüne *cut* schneiden Suchbäume weg, die keine Lösungen enthalten. Sie verändern die deklarative Bedeutung des Programms nicht und können weggelassen werden.
- rote *cut* schneiden Suchbäume weg, die Lösungen enthalten. Sie verändern die deklarative Bedeutung des Programms und können nicht weggelassen werden.

Der *cut*-Operator macht die Bedeutung eines Programms von der prozeduralen Interpretierung abhängig, somit wird die angestrebte Deklarativität verletzt.
Das $fail$-Prädikat ist ein häufig benutztes Prädikat zur Steuerung des Ablaufs eines Programms. $fail$ scheitert stets und erzwingt ein *Backtracking*. $true$ ist das Gegenstück zu fail und gelingt immer.

3.6 Negation

Prolog bietet dem Anwender das Prädikat not, um eine Negation von Aussagen oder Ziele auszudrücken. *not* sollte hierbei nicht als die logische Verneinung aufgefasst werden, sondern eher als Fehlschlagen einer Berechnung. Das Ziel

?- $not(X)$

gilt dann als bewiesen, wenn X fehlschlägt. Falls X allerdings herleitbar ist, dann schlägt $not(X)$ fehl. Not kann mit dem *cut*-Operator und mit dem $fail$ wie folgt implementiert werden:
Beispiel:

$not(X)$:-$X, !, fail.$
$not(X).$
?- $not(3 < 2).$
Yes
?- $not(2 < 3).$
No

Tritt eine ungebundene Variable im Argument von not auf, so führt dies auch hier zu Problemen.

3.7 Zusammenfassung

Prolog fehlt es an bestimmten Elementen zur Strukturierung, wie z. B. Module, um Daten von Programmen getrennt zu halten. Dies kann in größeren Projekten zu

Unübersichtlichkeit und Programmierfehlern führen. Weitere Schwierigkeiten entstehen bei arithmetischen Operationen, wenn in Ausdrücken ungebundene Variablen enthalten sind. Dies reduziert die Effizienz. Der Umgang mit dem *cut*-Operator ist mit Vorsicht zu benutzen, da unerwünschte Nebeneffekte auftreten. Die Anordnung der Programmklauseln sind für das Ergebnis von Bedeutung, denn das Tiefensuche-Verfahren kann durch unglückliche Anordnung in endlose Schleife geraten. Somit wird die Effizienz der Tiefensuchen-Verfahren mit der Unvollständigkeit erkauft.

Kapitel 4

Constraint Programmierung

Constraint Programmierung ist eine Softwaretechnologie, die es möglich macht, komplexe kombinatorische Probleme aus dem realen Leben auf eine natürliche Art zu formulieren und zu lösen. Sie stellt ein Mittel dar, eine Klasse von Problemen zu behandeln, die in der Theoretischen Informatik als schwierige Probleme (NP-Probleme) bezeichnet werden. Dabei nutzt Constraint Programmierung Konzepte von verschiedenen wissenschaftlichen Branchen und Richtungen, u.a. der Mathematik, der künstlichen Intelligenz und der formalen Sprachen.

4.1 Constraints

Das englische Wort Constraint bedeutet Rand- bzw. Nebenbedingung oder auch Einschränkung. Constraints eignen sich zur Darstellung von unvollständigen Informationen und zur Beschreibung der Eigenschaften und Beziehungen von teilweise unbekannten Objekten. Unter Constraint Programmierung versteht man die Formulierung von Aufgaben mit Hilfe von Constraints, die von einer Lösung erfüllt sein müssen. Ähnlich wie in der logischen Programmierung besteht die Lösung eines Problems überwiegend in der Problembeschreibung.

Vorteile der Nutzung von Constraints sind unter anderem ([Barták99]):

- Partielle Informationen können ausgedrückt und zur Lösung benutzt werden,
- Constraint sind heterogen, da sie Relationen zwischen Variablen über verschiedenen Wertebereichen darstellen, z.B. bei der Aussage:

  $$X = length(Y),$$

  dabei kann X für eine Zahl stehen und Y für eine Datenstruktur.
- Constraints sind ungerichtet und stellen dadurch mehr Informationen dar, z.B. die Beziehung

  $$X = Y + 2,$$

stellt die Relation (X, Y) sowie die Relation (Y, X), denn für Y gilt :

$$Y = X - 2,$$

- Constraints sind deklarative Art und beschreiben lediglich, welche Beziehungen zwischen Variablen gelten müssen und nicht wie diese vollbracht werden sollen.

Die deklarative Eigenschaft von Constraints macht Modellierung von Problemen weitgehend unabhängig von den Lösungsmethoden.

Marriott und Stucky geben in [KmPs98a] eine formale Definition für Constraints und ihrer Äquivalenz zueinander wie folgt:

Definition
Ein Constraint C ist der Form:

$$c_1 \wedge c_2 ... \wedge c_n, \text{ wobei } n \geq 0.$$

$c_1, ..., c_n$ werden Primitivconstraints genannt.
$C = (c_1 \wedge c_2 \wedge ... \wedge c_n)$ ist erfüllt genau dann, wenn alle Primitivconstraints erfüllt sind. Es existieren zwei ausgezeichnete Constraints, $wahr$ und $falsch$. Das leere Constraint $(n = 0)$ wird mit $wahr$ bezeichnet und gilt immer als erfüllt und das Constraint $falsch$ gilt als unerfüllbar.

Beispiel:

$(X > 5) \wedge (Y < X)$	erfüllbar
$(X > 4) \wedge (Y < 3) \wedge (Y > X)$	unerfüllbar

Definition
Zwei Constraints gelten als äquivalent genau dann, wenn sie dieselben Lösungsmengen besitzen, d.h. jede Belegung von Variablen, die das eine Constraint erfüllt, so erfüllt sie auch das andere Constraint.

Unter Stelligkeit eines Constraints verstehen wir die Anzahl der Variablen, die von dem Constraint betroffen sind. So ergeben sich folgende Constraintypen:

1. unäre Constraints beziehen sich nur auf eine Variable z. B.

   $$X < 4$$

2. binäre Constraints beziehen sich auf zwei Variablen z. B.

   $$X = 3 - Y$$

3. komplexe Constraints beziehen sich auf mehr als zwei Variablen z.B.

   $$X + Y + Z = 10$$

Es wurde gezeigt, dass komplexe Constraints in äquivalente binäre Constraints umgewandelt werden können [Barták99].

Constraints werden nach der Art der Ausdrucksform unterschieden. In [KmPs98a] wurden folgende und einige weitere Sorten von Constraints untersucht und beschrieben:

- arithmetische Constraints,
- Baum-Constraints (tree constraints),
- Boolesche Constraints (Variable können nur zwei Werte haben, 0 oder 1).

4.2 Constraint- Erfüllbarkeitsproblem

Ein Constraint-Erfüllbarkeitsproblem *(Constraint Satisfaction problem,CSP)* ist ein formales Modell, mit dem wir verschiedenartige Kombinatorische und Optimierungsprobleme in der realen Welt formulieren können. Formal kann CSP wie folgt beschrieben werden.

Definition
Ein CSP ist definiert als:

- eine Menge von Variablen $X = \{x_1, ..., x_n\}$,
- für jede Variable x_i eine Menge D_i von möglichen Werten, dabei können Werte numerisch oder symbolisch sein,
- eine Menge von Constraints über die Variablen.

Die Lösung eines CSP ist demnach eine Belegung aller Variablen, so dass alle Constraints erfüllt sind. Gesucht ist in der Regel

- eine Lösung,
- alle Lösungen oder
- optimale Lösung bezüglich einer Kostenfunktion, die für eine spezielle Problemvariable den bestmöglichen Wert aus dem Wertebereich der Variable zuweist. Die Suche nach der besten Lösung wird Constraint Optimierungsproblem genannt.

Das Constraint-Erfüllbarkeitsproblem gehört zu den NP-Problemen. Mit Constraint Programmierung werden einige dieser Probleme effektiv gelöst.

Ein typisches Constraint Programm beginnt mit der Angabe von Wertebereichen für die Problemvariablen und einzuhaltenden Constraints. Anschließend wird die Suche nach Belegungen gestartet. Wir wollen im Folgenden ein Beispiel [KmPs98a] für ein Constraint-Erfüllbarkeitsproblem angeben.

Map colouring (Färbung von Landkarten):
Problem:
Färbe die Regionen einer Karte mit drei Farben (rot, gelb, blau) so ein, dass keine zwei aneinandergrenzenden Regionen dieselbe Farbe haben.
Dieses Problem kann wie folgt als Constraint- Erfüllbarkeitsproblem modelliert werden:

- Die Bundesstaaten entsprechen den Variablen,
- die drei Farben sind der Wertebereich für alle Variablen und
- die Bedingung, dass keine benachbarten Staaten gleiche Farbe haben dürfen, können wir als Constraint auffassen und für alle benachbarten Staaten explizit angegeben.

$D(X)$ stellt den Wertebereich für Variable X dar. Die formale Modellierung des Problems ist:

$$D(\text{WA}) = D(\text{NT}) = D(\text{SA}) = D(\text{NSW}) =$$
$$D(\text{Q}) = D(\text{V}) = D(\text{T}) = \{\text{rot, gelb, blau}\}$$

$$(WA \neq NT) \wedge (WA \neq SA) \wedge (NT \neq SA) \wedge (SA \neq Q) \wedge (NT \neq Q) \wedge$$
$$(SA \neq NSW) \wedge (SA \neq V) \wedge (Q \neq NSW) \wedge (NSW \neq V)$$

Abbildung 4.1:

4.3 Constraint-Löser

Damit Constraintlösen in Computerprogramme integriert werden kann, und Constraints und ihre Lösungen einen Einfluss auf den Ablauf von Programmen haben können, muss es einen Programmteil geben, der Constraints verwaltet und löst. Wir nennen diesen Programmteil Constraint-Löser.

Die Erfüllbarkeitsfrage von Constraints wird durch den Constraint-Löser gelöst. Bei arithmetischen linearen Constraints, wie z.B.

$$X + Y = 7,\ X - Y = 3.$$

könnte der Constraint-Löser das Gaußsche Eliminationsverfahren anwenden, um sie zu lösen. Der Constraint-Löser soll dann die Gleichungen möglich soweit vereinfachen, dass Wertebelegungen für X und Y explizit werden:

$$X = 5,\ Y = 2.$$

Ein Constraint-Löser sammelt während des Programmablaufs ständig Constraints, vereinfacht oder löst sie und prüft, ob die Zusammensetzung der bisher bekannten Constraints konsistent ist, d.h. dass sich Constraints nicht gegenseitig ausschließen. Für das Vereinfachen von Constraints ist es notwendig Implikationen von Constraints zu definieren.

Definition
Wir sagen ein Constraint C impliziert $C'(C \rightarrow C')$ genau dann, wenn die Lösungsmenge von C eine Teilmenge der Lösungsmenge von C' ist, d.h. wenn Constraint C' redundant bezüglich C ist. Bei Vereinfachungen einer gegebenen Menge von Constraints werden redundante Primitivconstraints bezüglich der restlichen Constraints eliminiert.

Für den Constraintlöser gibt es folgende Arbeitsschritte:

- **Vereinfachung (*Simplification*)**

  Der Constraint-Löser versucht einen Constraint in ein, bezüglich einer gegebenen Menge von Variablen äquivalentes Primitivconstraint umzuwandeln.

  Es gilt beispielsweise folgendes:

  $$(X > 3) \rightarrow (X > 4)$$

  Wir können damit das Constraint $(X > 3 \wedge X > 4)$ vereinfachen zu $(X > 4)$, indem wir das redundante Primitivconstraint $(X > 3)$ eliminieren.

- **Feststellung (*Determination*)**

  Löser stellt fest, dass eine Variable in einem Constraint nur einen möglichen Wert haben kann. Diese Information kann dann an andere Constraints weitergegeben werden, so dass sie eventuell vereinfacht werden können. Beispiel:

  $$X \geq 2 \wedge 2 \geq X$$

  $X = 2$ ist die Feststellung in diesem Beispiel.

- **Projektion**

Wird ein Constraint auf eine Menge von Variablen projiziert, die in dem Constraint vorkommen, so können eventuell weitere Feststellungen und Vereinfachungen vorgenommen werden.

Beispiel: Für einen Constraint-Löser über der Domäne der reellen Zahlen ergibt Projektion von $(X < Y) \wedge (Y < Z)$ auf X und Z liefert $(X < Z)$.

Constraint-Löser unterscheiden sich bezüglich Wertebereiche für Variable und bezüglich Semantik und Syntax der Ausdrücke, mit denen Constraints formuliert werden können. Die Arbeitsweise eines Constraint-Lösers wird deshalb im Zusammenhang mit dem Wertebereich der Problemvariablen beschrieben. Es existieren Constraint-Löser u. a. für:

- Gleichungen zwischen Booleschen Termen,
- lineare Gleichungen und Ungleichungen über rationale bzw. reelle Zahlen sowie
- verschiedenartige Relationen über endliche Wertebereiche.

Constraint-Löser sollen bestimmte Eigenschaften besitzen, damit sie bei der Lösung von Problemen effektiv benutzt werden können.

Definition
Ein Constraint-Löser L für einen bestimmten Wertebereich D heißt vollständig, wenn L für jedes Constraint C in D entscheiden kann, ob C erfüllbar ist. [thom93]

Vollständigkeit gilt als die wichtigste Eigenschaft, die wir von einem Constraint-Löser fordern. Diese ist allerdings vom Wertebereich der Variable abhängig. Constraint-Löser über unendliche Wertebereiche können, bei der Suche nach einem Wert für eine Variable, in eine endlose Schleife geraten, so dass in endlicher Zeit nicht bestimmt werden kann, ob das Constraint erfüllbar ist. Unvollständige Constraint-Löser können allerdings nützlich sein. Bei der Benutzung unvollständiger Constraint-Löser erlauben wir, dass sie auch die Antwort „*unbekannt*" neben den beiden anderen Antworten *erfüllbar* und *unerfüllbar* liefern. Sie müssen allerdings auf die Erfüllbarkeitsfrage korrekt antworten, d.h. wenn der Löser als Ergebnis erfüllbar bzw. unerfüllbar liefert, dann muss dieses Ergebnis auch stimmen.

Folgende weitere Eigenschaften werden, nach [KmPs98a] , von einem Constraint-Löser gefordert:

- mengenorientiert (*set based*)

  Die Reihenfolge, in der Constraints angegeben werden, soll für die Lösung keine Rolle spielen. Dies gilt auch für die Anzahl, wie oft dasselbe Constraint angegeben wird.

- monoton (*monotonie*)

  Gilt das Constraint C als nicht erfüllbar, so ist jede Konjunktion($\wedge$) von C mit anderen Constraints auch unerfüllbar.

- variablenunabhängig

  Die Lösungsmenge von zwei, bis auf die Unbenennung der Variable äquivalenten Constraints soll dieselbe sein.

Constraint-Löser, die diese Eigenschaft erfüllen werden in [KmPs98a] „*well -behaved*" genannt. In den folgenden Kapiteln werden wir uns nur mit Constraint-Löser beschäftigen, die diese Eigenschaften besitzen.

4.4 Constraint Programmierung über endlichen Wertebereichen

Viele kombinatorische Probleme in der Industrie lassen sich als Constraint- Erfüllbarkeitsproblem (CSP) über endlichen Wertebereichen formulieren. Constraint-Löser über endliche Wertebereiche werden in der Praxis am meisten eingesetzt. Es gibt also nur endlich viele Werte, mit denen wir Problemvariable mit Werten belegen können. Dies hat zur Folge, dass es auch nur endlich viele Arten von Möglichkeiten gibt, alle Problemvariable mit Werten zu belegen.

Lösungen von CSP können auch als Suchprobleme bezeichnet werden. Die Menge aller möglichen Belegungen von Variablen mit Werten aus ihren Wertebereichen bildet dabei den Suchbaum. Der große Vorteil dieser Systeme gegenüber denen mit unendlichen Wertebereichen liegt daran, dass es viele vollständige Löser gibt.

Wir unterscheiden zwischen folgenden Vorgehensweisen bei der Suche nach Lösungen:

- Konsistenz-Techniken (unvollständige Suche),
- systematische Suche (vollständige Suche),
- Constraint Propagierung (*Constraint propagation*),
- stochastische und heuristische Algorithmen.

4.4.1 Konsistenz-Techniken

Mit Konsistenz-Techniken lässt sich herausfinden, welche Werte für Variable vernachlässigt werden können, ohne die Lösungsmenge zu verändern. Bevor mit der Belegung der Variable eines CSP begonnen wird, können Werte für bestimmte Variablen aus dem Wertebereich entfernt werden, die offensichtlich nicht zur Lösung des CSP beitragen können.

Zur Verdeutlichung der Aufgabestellung können CSP als Constraintnetze bzw. Constraintgraphen dargestellt werden. Dabei stehen die Knoten für die Variablen und die Kanten für die Constraints. Unäre Constraints sind die Kanten, die von einem Knoten zu sich selbst zeigen, und binäre Constraints entsprechen den Kanten zwischen zwei Knoten.

Beispiel:
Für das oben angegebene Map colouring- Problem kann das Constraintnetz wie folgt konstruiert werden:

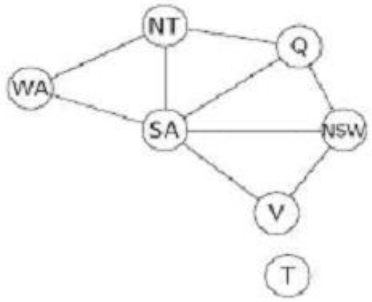

Abbildung 4.2:

Konsistenz wird erreicht, indem sukzessive alle Constraints betrachtet werden. Dabei werden alle Werte einer Variable eliminiert, die ein Constraint verletzen. In [Bartàk99] werden folgende Stufen der Konsistenz vorgestellt:

Definition
Knotenkonsistenz (*Node consistency*)
Ein Knoten mit Variable X ist konsistent, wenn jedes unäre Constraint C mit der Variable X von jeder Belegung aus dem aktuellen Wertebereich für X erfüllt wird. Ein Knoten mit Variable X ist konsistent, wenn jedes unäre Constraint C mit der Variable X von jeder Belegung aus dem aktuellen Wertebereich für X erfüllt wird.

Definition
Kantenkonsistenz: (*Arc Consistency*)
Eine (gerichtete) Kante $X \rightarrow Y$ ist konsistent, wenn zu jedem Wert aus dem aktuellen Wertebereich für X ein Wert aus dem aktuellen Wertebereich für Y existiert, so dass die entsprechenden Belegungen das dazugehörige binäre Constraint erfüllen. Sind in einem Constraintnetz alle Kanten konsistent, so sagen wir das CSP ist Kantenkonsistent.

Knotenkonsistenz kann mit einem Durchlauf über alle Knoten erreicht werden. Kantenkonsistenz erfordert im Allgemeinen mehrere Durchläufe. Eliminierung von Werten nach einmaliger Überprüfung der Kantenkonsistenz kann Konsequenzen für andere Variablen mit sich bringen. Daher ist es notwendig, die Kantenkonsistenzregel solange anzuwenden, bis bei einem Durchgang für keine Variable ein Wert entfernt wird. Je nach Methode welcher Constraint im nächsten Durchlauf auf Konsistenz geprüft wird, sind verschiede Erweiterungen der Kantenkonsistenz untersucht worden. z.B. AC3-7 [Tsang02].

Die Tatsache, dass ein Constraintnetz sowohl Knoten- als auch Kantenkonsistent ist, impliziert nicht, dass für das entsprechende System zwangsweise eine Lösung existiert.
Beispiel Das System

$$X \neq Y \wedge X \neq Z \wedge Y \neq Z \text{ mit}$$
$$D(X) = D(Y) = D(Z) = \{1, 2\}$$

ist kantenkonsistent, weil bei der Betrachtung der einzelnen Constraints für jede Belegung einer Variablen immer ein Wert der anderen Variablen gefunden werden kann. In seiner Gesamtheit besitzt das Netz aber keine Lösung.

Da Kantenkonsistenz nicht alle inkonsistenten Werte aus den Wertebereichen der Constraint- Variablen ausschließt, und zudem nicht sichergestellt ist, dass ein kantenkonsistentes Constraintnetz eine Lösung besitzt, wurden höhere Konsistenzgrade und Verfahren zur Konsistenzherstellung entwickelt. Die natürliche Erweiterung der Kantenkonsistenz ist die *Pfadkonsistenz.*

Definition
k-Konsistenz (*path consistency*):
Sei C ein Constraintnetz über einer Variablenmenge M und seien $\{x_1, ..., x_{k-1}\}$ Variablen eine Teilmenge von M, mit den Grundbereichen $D_1, ..., D_{k-1}$. Weiterhin seien die Werte $d_1, ..., d_{k-1}$ in $D_1, ..., D_{k-1}$ gegeben, so dass alle binäre Constraints zwischen den Variablen $x_1, ..., x_{k-1}$ erfüllt sind. Existiert nun für jede weitere Variable x_k mit Grundbereich D_k ein Wert d_k in D_k, so dass alle Constraints zwischen den k Variablen erfüllt sind, so heißt C k-Konsistent.

Definition
strenge k-Konsistenz (*bound consistency*):
Ein Constraintnetz heißt streng k konsistent, falls es für alle $j \leq k$ j-konsistent ist.

In einem streng $(n-1)$-konsistentem Constraintnetz auf n Variablen ist im Allgemeinen Suche nicht zu vermeiden. Wenn ein CSP mit n Variablen streng n-konsistent ist, dann ist Generierung einer konsistenten Lösung ohne Suche möglich. In diesem Zustand ist ein CSP global konsistent. Globale Konsistenz garantiert Konfliktfreiheit. Die Wertebereiche der Constraint-Variablen enthalten

ausschließlich Werte, die Teil einer Lösung sind. Somit lässt sich jede konsistente Teilbelegung der Constraint-Variablen zu einer vollständigen Lösung des CSP erweitern.

Je höher der Grad einer Konsistenzbedingung, desto mehr Inkonsistenzen können erkannt werden. Der Aufwand für Erstellung der Konsistenz nimmt jedoch mit der Anzahl der Variablen zu. Beim Einsatz von strengen k-Konsistenzen muss man also immer zusehen, ob sich der Aufwand lohnt.

4.4.2 Systematische Suche

Eine systematische Suche nach einer Lösung von CSP wird vollständig genannt, wenn sie den gesamten Suchraum abdeckt. Vollständige Suche garantiert den Erfolg, wenn es eine Lösung gibt. Dies ist insbesondere wichtig, wenn nach der besten bzw. optimalen Lösung gesucht wird (Optimierungsprobleme). Auch unvollständige systematische Suche kann sinnvoll eingesetzt werden, wenn z.B. nur nach einer beliebigen konsistenten Belegung von Variablen gesucht wird. Wir unterscheiden zusätzlich zwischen konstruktiven und „verbesserungsorientierten" (*move based*) Suchen. Bei der konstruktiven Suche werden Variable nacheinander mit Werten belegt, wobei bei der verbesserungsorientierten Suche von einer kompletten Zuordnung der Variable mit Werten in eine andere komplette Zuordnung gewechselt wird.

Generieren und Testen
Die trivialste Art die Erfüllbarkeitsfrage für Constraints zu lösen besteht darin, alle möglichen Belegungen der Problemvariablen zu untersuchen, bis eine Lösung gefunden wird. Zunächst wird eine komplette Belegung generiert und getestet, ob alle Constraints erfüllt sind. Ist dies nicht der Fall, so werden systematisch alle weiteren Belegungen generiert und getestet. Wird dabei eine Lösung gefunden, so terminiert der Algorithmus und liefert die Lösung. Anderenfalls muss der gesamte Suchraum durchsucht werden. Dies erfordert bekanntlich erheblich viel Zeit, da der Suchraum exponentiell mit der Problemgröße wächst.

Die Lösungsmethode „*generate and test*" ist nicht effizient, da erheblich viele unnötige Belegungen generiert werden, die erst in der Testphase als ungültig erkannt werden. Constraints werden folglich nur passiv benutzt, da die Inkonsistenz von Constraints spät erkannt wird. Die folgende Methode gilt als deutlich effizienter.

Backtracking
Das *Backtracking*-Verfahren versucht, wie wir bereits im vorherigen Kapitel erwähnt haben, Teillösungen zu kompletten Lösungen Schritt für Schritt zu ergänzen, indem Variable sequenziell mit Werten belegt und anschließend mit den bereits belegten auf Inkonsistenz getestet werden. Sind alle Variablen eines Constraints belegt und die Constraints nicht verletzt, so ist eine Lösung gefunden. Es gibt keine Lösung, wenn alle Belegungen ohne Erfolg untersucht worden sind.

Wir wollen zur Verdeutlichung des einfachen Backtracking folgendes Beispiel [KmPs98a] angeben:

Deklaration der Wertebereiche: D(X) = D(Y) = D(Z) = $\{1, 2\}$ Angabe von Constraint : $(X < Y \wedge Y < Z)$

Die Suche des *Backtracking*-Verfahren kann als ein Baum dargestellt werden (Siehe Abb. 3). Jedes Blatt entspricht dabei einer der kompletten Belegungen der Variablenmenge, und jede Stufe des Teilbaums entspricht einer Entscheidung eine der Variablen mit einem Wert zu belegen. Je nach Entscheidung, welche Variable mit welchem Wert zuerst belegt wird, ergeben sich unterschiedliche Baumstrukturen. Das oben angegebene Beispiel ist bekanntlich unlösbar. *Backtracking* probiert allerdings alle Belegungen aus, um dies festzustellen.

Backtracking Algorithmen haben eine Laufzeit von $O(d^n)$ bei n Variablen und maximaler Wertebereichgröße d. Deshalb sind für die Praxis Verbesserungen der exponentiellen Laufzeit notwendig.

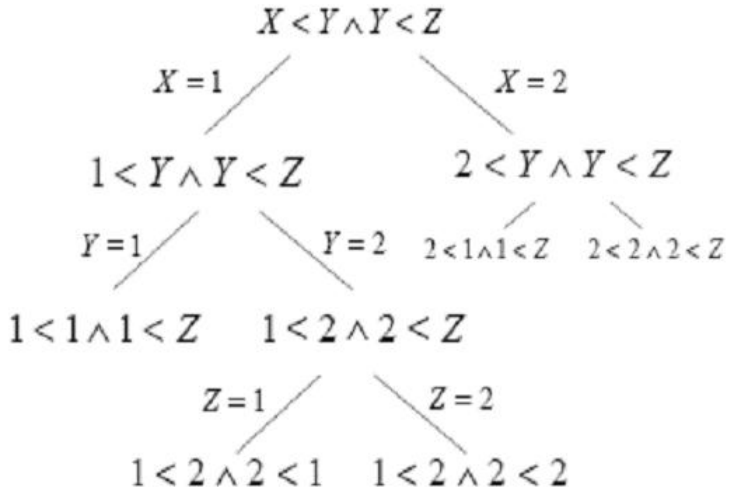

Abbildung 4.3:

4.4.3 Constraint Propagierung (*Constraint propagation*)

Im Allgemeinen reichen Konsistenz-Techniken nicht aus, um Probleme vollständig zu lösen. Die Integration von der vollständigen aber uneffizienten systematischen Suche und unvollständigen aber effizienten Konsistenzprüfungen hat sich als bessere Methode erwiesen. Das Ziel ist dabei die Vermeidung zukünftiger Konflikte im Voraus,

anstatt sie erst beim Auftreten zu beseitigen, wie es beim *Backtracking*-Verfahren der Fall ist.

In [Bartak1] wird zwischen vorwärts- und rückwärtsorientierten Propagierungsmethoden unterschieden. Wir wollen im Folgenden jeweils eine Variante der beiden Methoden erläutern.

Das Vorwärtsschema (*Look ahead scheme*)
Vorwärtstest (*Forward Checking*):
Das Effizienzproblem des zu späten Erkennens von Konflikten kann mit einem Vorwärtstest behandelt werden. Wann immer eine Variable X mit einem Wert belegt wird, werden für alle, noch nicht belegten und mit X durch einen Constraint verbundenen, Variablen diejenigen Werte aus den Wertebereichen eliminiert, die zu Inkonsistenzen führen. Wird dabei einer der Wertebereiche leer, so wird zurückgesetzt. Bei Belegung der j-ten Variablen mit einem Wert werden inkonsistente Werte für folgende Variablen eliminiert.
Es gibt weitere Variationen der *Look-ahead*-Methode, wie z.B. *Partial Look-ahead*, *Full Look-ahead* und *Maintaining Arc Consistency*.

Das Rückwärtsschema (*Look back scheme*)
Zurückspringen (*Backjumping*):
Optimierung von Backtracking kann durch konfliktgesteuertem- anstelle von chronologischem Rücksetzen erreicht werden, indem bei Inkonsistenz nicht auf die letzte Variablenbelegung zurückgesprungen wird, sondern auf die Belegung, welche die Inkonsistenz verursacht hat. Es gibt verschiedene Variationen der *Backjumping*-Methode z.B. *Backchecking* und *Backmarking*.
Im Allgemeinen erkennen jedoch alle Variationen des Rückwärtsschemas Inkonsistenz spät, da sie die Inkonsistenzen nicht im Voraus vermeiden.

4.4.4 Stochastische Suche und Heuristiken

Stochastische Suchverfahren sind unvollständig und betrachten das CSP anhand lokaler Kriterien. Sie garantieren nicht, dass eine gefundene Lösung globale Gültigkeit besitzt. Sie werden oftmals auch lokale Suchverfahren (*local search*) genannt.

Stochastische Suchverfahren beginnen mit einer zufälligen kompletten Belegung aller Variablen. Es folgen Veränderungen aufgrund von lokalen Kriterien und Zufallswerten. Schritt für Schritt wird diese Wertezuweisung verbessert, bis keine oder nur noch möglichst wenig Konflikte vorliegen. Stochastische Verfahren finden häufig auch in sehr großen Suchräumen relativ schnell eine Lösung. Beispiele für bekannte Ansätze sind *Hill Climbing*, *Simulated Annealing*, Tabu-Suche und Genetische Algorithmen. Für bestimmte Anwendungen, bei denen frühzeitige Findung einer korrekten Lösung nützlicher ist als das Finden der besten Lösung, können unvollständige Suchverfahren sinnvoll sein, da vollständige Suchverfahren in der Regel mehr Zeit in Anspruch nehmen.

Im Kontext von allgemeinen Suchverfahren werden Heuristiken als problemspezifische und strategische Schätzungen zur Findung einer Lösung verstanden, mit dem Ziel eine, bezüglich der Laufzeit, optimale Reihenfolge der Arbeitsschritte zu finden.
Wir unterscheiden zwischen zwei Typen von Heuristiken:

Variablen-Ordnungsheuristiken
Die Belegung von Variablen mit Werten soll in einer Reihenfolge vorgenommen werden, so dass eine vollständige und konsistente Belegung mit möglichst wenig Backtracking hergestellt werden kann. Um zu verhindern, dass große Teile von bereits belegten Variablen aufgrund von erst spät festgestellten Inkonsistenzen wiederholt neu belegt werden müssen (*trashing*), ist es sinnvoll kritische Variable bereits zu Beginn einer Suche zu belegen. Eine Variable heißt kritisch, wenn von ihr viele andere Belegungen abhängig sind.

Variablen-Ordnungsheuristiken legen fest, welche Variable als nächstes mit einem Wert aus ihrer Domäne belegt werden soll. Dazu gibt es u. a. folgende Vorgehensweisen:

- *first fail*

  Diese Heuristik erstellt eine Belegungsreihenfolge so, dass Bereiche des Suchbaums, in denen ein Fehlschlagen wahrscheinlicher ist als in anderen, zuerst durchsucht werden. Mit dieser Heuristik kann man Inkonsistenz schneller erkennen und dadurch die Suche beschleunigen.

  Minimum remaining values (MRV):

  Bei der Zuordnung von Variablen mit Werten ist die Variable vorzuziehen, die die wenigsten zulässigen Werte besitzt.

  Maximum-Degree-Heuristic:

  Bei Zuordnung von Variablen mit Werten ist die Variable als nächstes zu belegen, die in den meisten Constraints mit unbelegten Variablen vorkommt.

Werte-Ordnungsheuristiken
Haben wir uns bereits für eine Variablenbelegung entschieden, so legen die Werte-Ordnungsheuristiken fest, welcher Wert zuerst ausgewählt wird.

- *least constraining value*

  Für eine Variable ist der Wert auszuwählen, der die wenigsten Werte für benachbarte Variablen im Constraintgraphen ausschließt.

Variablen- und Werte-Ordunsheuristiken sind unabhängig voneinander und ergänzen sich gegenseitig, deshalb können sie bei der Lösung von CSP gemeinsam verwendet werden.

4.5 Constraint-Optimierung

Bis jetzt haben wir bei Constraint-Erfüllbarkeitsproblem nur nach einer beliebigen Lösung gesucht, bei denen alle Constraints erfüllt sind. Oft sind Probleme jedoch der Art, dass man nach der besten Lösung unter den Möglichen Kandidaten sucht. Die Suche nach der besten Lösung wird Constraint-Optimierung genannt. Dies erfordert ein Maß, um vergleiche zwischen Lösungen durchzuführen. Es wird eine Kostenfunktion definiert, welche jede Lösung auf eine reelle Zahl abbildet. Gesucht ist also eine Lösung mit, je nach Problemsituation, minimalen oder maximalen Wert für die Kostfunktion. Ein Constraint-Optimierungsproblem wird formal wie folgt definiert [KmPs98a] :

Definition
Ein Constraint-Optimierungsproblem ist ein Tupel (C, f), wobei C ein Constraint ist und f eine Kostenfunktion über die Variablen des Constraints C auf eine reelle Zahl. Es wird immer nach einer Lösung mit minimalen Wert gesucht, denn man kann nach dem minimalen Wert für $-f$ suchen.

Wir sagen eine Lösung L wird „bevorzugt" von einer Lösung L'' genau dann, wenn

$$f(L) < f(L'').$$

Eine Lösung L ist eine optimale Lösung, wenn es keine Lösung L'' gibt, mit:

$$f(L'') < f(L).$$

Es ist allerdings nicht immer so, dass für solche Probleme eine optimale Lösung existiert, bei der die obige Bedingung erfüllt ist. Es gibt Situationen wo es keine optimale Lösung gibt, weil man zu jeder gefundenen Lösung eine bessere, bezüglich der Kostenfunktion, angeben kann.

Für das Constraint-Optimierungsproblem $(X < 0, f)$ mit $f(X) = r$, wobei r eine negative Zahl ist, findet man zu jeder Lösung eine bessere, denn zu jeder reellen Zahl kann eine kleinere gefunden werden.

In [KmPs98a] werden die so genannten *simplex*-Algorithmen zur Optimierung vorgestellt, die lineare arithmetische Gleichungen als Constraints behandeln. In den kommenden Kapiteln werden wir Optimierungsprobleme näher behandeln, in dem wir die Techniken der Constraint-Programmierung in Kombination mit den Techniken der logischen-Programmierung anwenden.

Kapitel 5

Constraint-logische Programmierung und ECL^{i}PS^e

Techniken der Constraint Programmierung können prinzipiell in verschiedenartigen Programmiersprachen integriert werden. Constraint-logische Programmierung (CLP) ist eine Erweiterung der logischen Programmierung um die effiziente Verarbeitung von Constraints.

Mit dieser Kombination wurde es möglich schnell und elegant komplexe kombinatorische Probleme zu lösen, u.a. in den Gebieten Produktions- und Ressourcenplanung, Transport-Optimierung, Personalplanung. Wie wir in Kapitel 3 bereits erwähnt haben, werden Objekte (Terme) der logischen Programmierung nicht interpretiert, und Gleichheit zweier Terme bedeutet lediglich syntaktische Äquivalenz der Terme. Die Bedeutung von Objekten wird dadurch oft nur mittels benutzerdefinierter Klauseln dargestellt. Diese Probleme werden mit Hilfe von Constraints gelöst.

Die Idee der Constraint-logischen Programmierung ist also, dass bestimmte Prädikate als Constraint deklarativ formuliert und effizient gelöst werden können. Dabei wird die rein syntaktische Gleichheit der logischen Programmiersprachen so erweitert, dass die Lösung eines Zieles, statt mittels SLD-Resolution, durch eine Kombination von SLD-Resolution und Constraint-Löser erfolgt.

Logische Programme durchsuchen in der Regel, wenn nicht vom Programmierer mittels Kontrollstrukturen beeinträchtigt, den gesamten Suchraum des entsprechenden Zieles. Durch die Integration von Constraints werden Propagierungs-Techniken der Constraint-Programmierung benutzt, um bei der Suche Teilgebiete im Voraus auszuschließen, in denen keine Lösungen zu finden sind. Die Suche nach Lösungen wird zusätzlich durch heuristikbasierte Verfahren unterstützt, die spezifische Information aus Anwendungsgebieten ausnutzen.

Nach [ThSl99] ist die logische Programmierung als Spezialfall der CLP zu betrachten, wobei das einzige Constraint die syntaktische Gleichheit zwischen Termen ist, und der Unifikationsalgorithmus zur Lösung solcher Constraints

benutzt wird. Logische Programme durchsuchen in der Regel, wenn nicht von Programmierer mittels Kontrollstrukturen beeinträchtigt, den gesamten Suchraum des entsprechenden Zieles. Durch die Integration von Constraints werden Propagierungs-Techniken der Constraint-Programmierung benutzt, um bei der Suche Teilgebiete im Voraus auszuschließen, in denen keine Lösungen zu finden sind. Die Suche nach Lösungen wird zusätzlich durch heuristikbasierte Verfahren unterstützt, die spezifische Information aus Anwendungsgebieten ausnutzen.

5.1 CLP-Paradigma

Jeder Ansatz der CLP wird durch die zugrundeliegende Domäne, den benutzten Constraint-Löser und die vom Löser benutzten Propagierungs-Techniken eindeutig gekennzeichnet. In diesem Abschnitt wollen wir allerdings einen allgemeinen Ansatz für Constraint-logische Programmierung mit Hilfe eines CLP-Paradigmas darstellen. Das CLP-Paradigma wird ohne Angabe der Domäne erläutert, in der Constraints ausgerückt werden. Mit der Festlegung der Domäne entstehen verschiedene Sprachen. Die Ausdruckstärke einer solchen CLP-Sprache wird an der Funktionalität und Qualität des Constraint-Lösers gemessen. Alle diese Sprachen besitzen die gemeinsame Eigenschaft, dass sie bei einem Ziel die von Prolog bekannte Auswertungsmethode benutzen. Dies entspricht dem Tiefensuche-Verfahren in Kombination mit *Backtracking*, wobei der dem Ziel entsprechende SLD-Baum von, links nach rechts durchlaufen wird.

Wir beginnen mit einer allgemeinen Definition für die Syntax von CLP-Sprachen. Wir nehmen dabei an, dass Constraint-Löser die im letzten Kapitel geforderten Eigenschaften besitzen, nämlich, dass sie monoton, mengenorientiert und variablenunabhängig sind.

Wir beginnen mit einer allgemeinen Definition für die Syntax von CLP-Sprachen, wie sie in [KmPs98a] zu finden ist. Wir nehmen dabei an, dass Constraint-Löser die im letzten Kapitel geforderten Eigenschaften besitzen, nämlich, dass sie monoton, mengenorientiert und variablenunabhängig sind.

5.1.1 CLP-Syntax

Definition
Ein Constraintatom ist der Form $p(t_1, ..., t_n)$, wobei p ein n-stelliges Prädikatensymbol ist und $t_1, ..., t_n$ sind gültige Ausdrücke über Domainvariable.
Ein Constraintatom ist entweder ein Primitivconstraint oder benutzerdefiniertes Constraint.
Ein Ziel L ist eine Folge von Constraintatomen der Form:

$$L_1, L_2, L_3, ..., L_n, \text{ wobei } n >= 0 \text{ ist.}$$

Ist $n = 0$, so heißt G das leere Ziel und wird mit $\circ$ bezeichnet.

Eine Regel R ist der Form A :- B, wobei A ein benutzter definiertes Constraint ist und B ein Ziel. Analog zu Logik werden A und B als der Kopf bzw. der Rumpf der Regel R bezeichnet.
Ein Fakt ist eine Regel mit dem leeren Ziel als Kopf.
Ein Constraint-logisches Programm ist eine Folge von Regeln.

In einem Constraint-logischen Programm können Constraints mehrere Definitionen haben. Dies gibt dem Programmierer die Möglichkeit, Fallunterscheidungen bei der Problemmodellierung zu realisieren. Auch rekursive Definitionen können ähnlich wie in Prolog durch Angabe zweier Definitionen realisiert werden.

5.1.2 Auswertung von CLP-Programmen

Die Auswertung eines CLP-Programms entspricht einer Ableitung des Ziels. In jeder Phase der Ableitung befindet sich die Auswertung in einem bestimmten Zustand, der als eine Konjunktion von Primitivconstraints und eine Konjunktion der noch zu bearbeitenden Ziele eindeutig definiert wird.

Definition
Ein Zustand ist ein Paar $[G|C]$, wobei G ein Ziel und C ein Constraint ist. G wird mit Ziel-Speicher bezeichnet und C mit Constraint-Speicher.
Ein Anfangzustand ist von der Form $[G|true]$:

1. Ist das Zielatom ein Primitivconstraint, dann wird es in den Constraint-Speicher aufgenommen und der dadurch veränderte Speicher wird auf Erfüllbarkeit getestet. Bei Unerfüllbarkeit bricht das Verfahren ab, da es, wegen der Monotonie des Constraint-Lösers, keine Lösung mehr geben kann. Anderenfalls werden die im letzten Kapitel vorgestellten Schritte, nämlich Vereinfachung, Feststellung und Projektion, vorgenommen, bis keine weiteren Vereinfachungen möglich sind.

2. Ist das Zielatom ein benutzerdefiniertes Constraint, so wird ein passendes benutzerdefiniertes Constraint gesucht, mit dem das Ziel unifiziert werden kann. Wird kein passendes Constraint gefunden, bricht das Verfahren ab und die Berechnung ist somit gescheitert. Anderenfalls wird das passende Constraint mit dem Zielatom resolviert und Teilziele werden weiter bearbeitet. Sind alle Ziele abgearbeitet, so ist der Inhalt des Constraint-Speichers als Antwort auszugeben. Dies kann unter Umständen eine indefinite Antwort sein, da es nicht immer möglich ist, Variablen mit festen Werten zu belegen.

Definition Ein Zustand $[G|C]$ heißt erfolgreicher Zustand, wenn G das leere Ziel und C nicht unerfüllbar ist.
Ein Zustand $[G|C]$ heißt erfolgloser Zustand ist, wenn G das leere Ziel ist und C unerfüllbar.

Eine Ableitung von $[G_0|C_0] \Rightarrow [G_n|C_n]$ heißt eine erfolgreiche Ableitung, wenn $[G_n|C_n]$ ein erfolgreicher Zustand ist. Sie heißt erfolglos, wenn $[G_n|C_n]$ ein erfolgloser Zustand ist.

Um Verwechslungen zu vermeiden, bezeichnen wir das leere Ziel bei einer erfolgreichen Ableitung mit $\circ$ und bei einer erfolglosen Ableitung mit $\bullet$.
Auf Grund der Tatsache, dass es mehrere Möglichkeiten gibt ein benutzerdefiniertes Constraint zu ersetzen, kann es für ein Ziel mehrere Ableitungen geben. Die Menge aller Ableitungen für ein Ziel bildet den Ableitungsbaum. Dabei entsprechen die Knoten den Zuständen und die Wurzel dem Anfangszustand.

Definition
Ein Ziel heißt endlich gescheitert, wenn es einen endlichen Ableitungsbaum besitzt, in dem alle Ableitungen erfolglos sind.

Die Berechnung eines Zieles einer CLP-Sprache, entspricht demnach dem Durchlauf durch den Ableitungsbaum des Zieles. Ist dabei ein erfolgreicher Zustand erreicht, so gibt das System dem Benutzer die Ergebnisse. Möchte der Benutzer weitere Ergebnisse, so kann weiter im Suchbaum nach erfolgreichen Zuständen gesucht werden. Ist ein erfolgloser Zustand erreicht und der Baum ist vollständig durchsucht, so gibt das System die Antwort *No*.

Wir wollen an dieser Stelle ein Beispiel einführen, um die Auswertungs- methode einer CLP Sprache mit der Menge der reellen Zahlen als Domäne und mit linearen und arithmetischen Gleichungen als Constraints erläutern. Das Gaußsche Eliminationsverfahren kann in diesem Fall vom Constraint- Löser verwendet werden, um Primitivconstraints Vereinfachen.

Problemstellung: Eine Mahlzeit, bestehend aus Vorspeise A und Hauptspeise M, ist eine leichte Mahlzeit, genau dann, wenn A und M jeweils einen positiven Kaloriewert I bzw. J besitzen, mit $I + J < 8$. Eine Liste von Kaloriewerten der verfügbaren Bestandteile von Mahlzeiten ist gegeben. Finde eine Zusammensetzung für eine leichte Mahlzeit aus den Bestandteilen, so dass der gewünschte Kaloriewert erreicht ist.
Die Problemstellung kann durch ein benutzerdefiniertes Constraint und Fakten wie folgt modelliert werden:

Ein CLP-Programm:

$$leichte\_Mahlzeit(A, M) \text{ :-}$$
$$I > 0, J > 0, I + J < 8,$$
$$vorspeise(A, I),$$
$$hauptspeise(M, J).$$

$vorspeise(salat, 5).$
$vorspeise(pasta, 3).$
$hauptspeise(fleisch, 6).$
$hauptspeise(fisch, 3).$

Die Kalorienwerten der Bestandteile werden als Fakten formuliert Wir zeigen nun die ersten Ableitungsschritte für das Ziel $leichte\_Mahlzeit(A, M)$, die zu einem Erfolglosenzustand führen.

Erfolglose Ableitung

Anfangszustand: $[leichte\_Mahlzeit(A, M)|true] \Rightarrow$

$[vorspeise(A, I), hauptspeise(M, J)|I > 0, J > 0, I + J < 8] \Rightarrow$

$[hauptspeise(M, J)|A = salat, I = 5, J > 0, 5 + J < 8] \Rightarrow$

$[\bullet|M = fleisch, J = 6A = salat, I = 5, J = 6, 5 + 6 < 8]$

Antwort: $M = fleisch, A = salat$

Primitivconstraints werden im Constraint-Speicher aufgenommen (nach dem Trennzeichen „|") und anschließend wird der Constraint-Speicher auf Konsistenz getestet. Dies gelingt und im nächsten Schritt wird aus dem Ziel-Speicher das am weitesten links stehende Teilziel ($vorspeise(A, I)$) reduziert. Die dabei entstehenden neuen Primitivconstraints werden aufgenommen. Der Constraint-Löser stellt Inkonsistenz im Constraint-Speicher fest und damit ist ein erfolglose Zustand erreicht. Dieser Zustand entspricht dem am weitesten links liegenden Blatt im folgenden Ableitungsbaum des Zieles. Die rot markierten Primitivconstraints sind die Inkonsistenzen

Die Wurzel im obigen Ableitungsbaum entspricht dem Ziel. Die Nummerierung der Knoten steht für die Reihenfolge, in der Schritt für Schritt der Ableitungsbaum durchsucht wird. Nach dem ersten erfolglosen Zustand (4) wird zum letzten Zustand (3) zurückgesetzt, wo es alternative Ableitungen gab. Die Auswertung läuft bis zum ersten erfolgreichen Zustand (8). Der Ableitungsbaum wird dadurch vollständig durchsucht. Der Inhalt der Constraint-Speicher wird als Antwort dem Benutzer zurückgegeben.
Die Reihenfolge der Ziele und der Teilziele im Rumpf eines benutzer- definierten Constraint sind für die Effizienz von CLP-Sprachen entscheidend. In dem obigen Beispiel könnten die Fakten anders sortiert werden, wodurch ein erfolgreicher Zustand erreicht wird bevor der Baum vollständig durchsucht worden ist. In diesem Fall gibt das System den aktuellen Constraint-Speicher als Antwort zurück und bietet dem Benutzer die Möglichkeit, den Baum weiter zu durchsuchen. Wird dies vom Benutzer bestätigt, so versucht das System weitere Antworten zu suchen. Dies gelingt jedoch nicht, da für das Ziel nur eine erfolgreiche Ableitung existiert. Die Antwort eines Zieles ändert sich also nicht durch die Änderung der Reihenfolge.

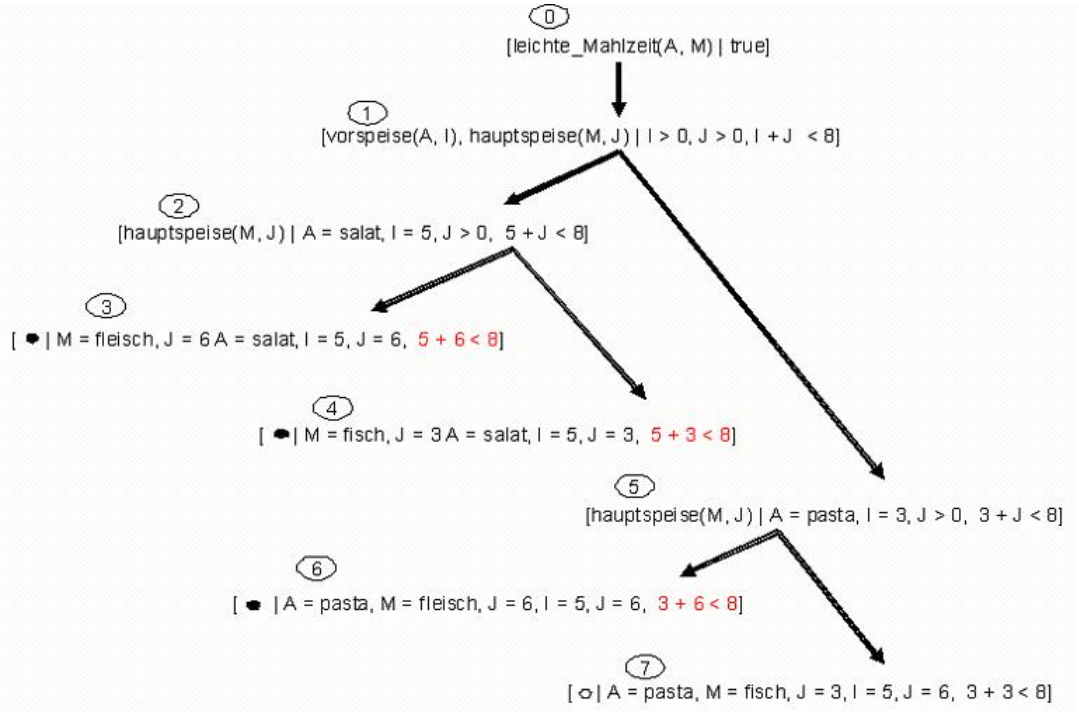

Abbildung 5.1:

Die Reihenfolge der benutzerdefinierten Constraints dagegen kann zu verschiedenen Antworten führen. CLP-Sprachen sind demnach nichtdeterministisch. Betrachten wir das folgende Beispiel:

$$finde(X) \text{ :- } finde(X), X > 0$$
$$finde(1).$$

Für das Ziel $finde(T)$ würde das System nicht terminieren, obwohl die Verstauchung der Constraints zu dem erfolgreichen Zustand $[\circ|T = 1]$ führen. Bei einem rekursiven benutzerdefinierten Constraint in einem CLP-Programm soll also, ähnlich wie in Prolog, die Abbruchbedingung ($finde(1).$) vor dem rekursiven Teil definiert werden. Andernfalls besitzt der Ableitungsbaum eine unendliche Ableitung. Es hat sich als nützlich erwiesen redundante Constraints einzuführen, um die Effizienz eines CLP-Programms zu steigern. Unter einem redundanten Constraint versteht man, ein Constraint, das aus einem Programm entfernt werden kann, ohne sich die Lösung eines beliebigen Zieles ändert. Durch redundante Constraints können die Länge der Ableitungsschritte reduziert werden.

5.2 ECL^{i}PS^e

ECL^{i}PS^e ist eine Constraint-logische Programmiersprache und besitzt die selbe Syntax wie Prolog. Sie wurde am Anfang vom *European Computer-Industry Research Centre (ECRC)* in München entwickelt. Dann wurde sie vom *Centre for Planning and Resource Control (IC-Parc)* des Imperial College in London übernommen und weiter entwickelt. Im Laufe der Diplomarbeit hat sich der Sitz jedoch erneut geändert.

ECL^{i}PS^e wird von *IC-Parc* für akademische Zwecke frei zur Verfügung gestellt. So auch für diese Arbeit. Aktuelle Information findet man auf der neuen Website von ECL^{i}PS^e:

http://eclipse.crosscoreop.com/eclipse/

ECL^{i}PS^e ist entwickelt mit dem Ziel Techniken verschiedener wissenschaftlicher Gebiete mit der Constraint-logischen Programmierung zu Kombinieren, so dass hybride Algorithmen (Constraint-Löser) im Zusammenspiel bei der Behandlung verschiedener u.a. kombinatorischer, scheduling und mathematischer Probleme eingesetzt werden können. Die Integration hybride Algorithmen erlaubt schnelle Prototyp-Entwicklungen und somit das Experimentieren mit alternativen Lösungsmethoden.

Die eingebaute Arbeitsweise von ECL^{i}PS^e entspricht der des oben beschrieben CLP-Paradigma. Dies wird allerdings vom Programmierer durch Prädikate, bei der Suche nach Lösungen beeinflusst wird. ECL^{i}PS^e kann in anderen Systemen eingebetet werden u.a. in Java, C, C++. Auvh andere Systeme können ins ECL^{i}PS^e System als Bibliotheken eingebette werden.

Anders als in Prolog werden in ECL^{i}PS^e Argumente von Strukturen benannt, so dass auf einzelne Argumente zugegriffen werden kann. Als Datenstrukturen in ECL^{i}PS^e gibt es u. a. Listen, Hash-Tabellen, Arrays.

Die verschieden Algorithmen in ECL^{i}PS^e können nach Verwendungszweck in zwei Gruppen eingeteilt werden:

- Verwaltung von Constraints

  Hier sind verschiedene Constraintlöser als Bibliotheken vorhanden, die in einem Programm gleichzeitig benutzt werden können.

- Suche nach Lösungen

  Hier sind alle Suchverfahren die wir in Kapitel 4 erläutert haben, als Constraints realisiert. (vollständige- und unvollständige- sowie konstruktive- und verbesserungsorientierte Suche).

In ECL^{i}PS^e sind alle Constraint-Löser in Form von Bibliotheken realisiert. Wenn in Programmen die Nutzung bestimmter Consrtaint-Löser nicht erwünscht ist, so ist das System im eingebauten Modus (*suspend-Modus*). In diesem Modus werden Ziele, bei denen für bestimmte Constraints die entsprechenden Variablen nicht gebunden sind, vorerst zurückgestellt. Sie werden zunächst als erfüllt angenommen und weitere Teilziele werden bearbeitet. Verzögerte Ziele gelten als „schlafend" und werden „geweckt", wenn die entsprechenden ungebundenen Variablen modifiziert werden.

Beispiel:
Für das folgende Ziel gilt mit dem Prädikat $=< /2$ antwortet das System wie folgt:

$?- X =< 4, X >= 3.$
$X = X$
There are 2 delayed goals.
Yes (0.00s cpu)

ECL^{i}PS^e enthält eine Fülle von Bibliotheken, die in Programmen geladen und verwendet werden können. Eine dieser Bibliotheken ist die *IC*-Bibliotek (*IC-ibrary*).

Wir wollen im Folgenden die *IC*-Bibliothek erläutern, da sie bei der Simulation der Prüfungsordnung in dieser Arbeit eingesetzt wird.

5.2.1 Der Interval Constraint-Löser (*IC-Solver*)

ECL^{i}PS^e unterstützt Constraint-logische Programmierung über endliche Wertebereiche mit der *IC*-Bibliothek. In dieser Bibliothek befindet sich u.a. der *IC*-Löser über eine endliche Menge ganzer und reeller Zahlen. Hier können Variablen beider Typen in Constraints gemischt benutzt werden. Dabei wird jede Zahl als ein Intervall dargestellt. Die Nutzung von Intervallen für reelle Zahlen ermöglicht eine bessere Abschätzung der Ungenauigkeiten bei Rechnungen, als dies mit Fließkommazahlen der Fall ist.

Eine Menge von Constraints können von *IC*-Löser verwaltet werden, u.a.folgende:

Domain-Constraints
Das Prädikat (:: /2)

$$Vars :: Dom$$

bedeutet, die Variable$Vars$ ist (bzw. die Variablen in der Liste $Vars$ sind) aus dem Wertebereich Dom.

Beispiel:

$$X :: 0..99,$$
$$[Z1, Z2] :: 0..9$$

Arithmetische Constraints
Folgende arithmetische Constraints können verwendet werden.

$T1\# < T2$, $T1\# > T2$, $T1\# = T2$, $T1\# <= T2$, $T1\# >= T2$, $T1\#\#T2$

Beispiel:

$$[X, Y, Z] :: 1..10,$$
$$2 * X + 3 * Y + 2\# < Z.$$

Globale Constraints
Globale Constraints können verwendet werden, um Aussagen über eine Menge von Objekten zu formulieren. Sie tragen zusätzlich zur besseren Lesbarkeit bei.

Beispiel:
Das Prädikat

$$alldiffernt(L)$$

bedeutet, dass alle Elemente in der Liste L paarweise verschieden sind.

Konsistenz-Techniken, Propagierungstechniken, und Suchmethoden, die wir im letzten Kapitel erläutert haben, werden in dem *IC*-Solver mittels Prädikate realisiert.

Ein Beispiel für Kantenkonsistenz ist das Prädikat $\#\backslash = /2$, bei dem Werte für Variablen entfernt werden, obwohl keine Variable gebunden ist.

?- $[X, Y] :: 1..5, X\# < Y.$
$X = X\{1..4\}$
$Y = Y\{2..5\}$
There is 1 delayed goal.
Yes(0.00s cpu)

Ein Beispiel für die Propagierungstechnik Vorwärtsschema ist das Prädikat $\#\backslash = /2$, bei dem Werte aus der Domäne der Variablen entfernt werden, bevor alle Variablen gebunden sind.

?-$X :: 1..5, X\#\backslash = 3.$
$X = X\{[1, 2, 4, 5]\}$
Yes(0.00s cpu)

5.2.2 Programmierung mit ECL^{i}PS^e

Programmierung mit ECL^{i}PS^e ist entweder durch eine graphische Benutzeroberfläche (*tk-ECL^{i}PS^e*) oder über den Kommando-Zeileninterpreter möglich. In ECL^{i}PS^e wird kein Editor zum Schreiben von Programmen mitgeliefert. Es kann jedoch ein beliebiger Text-Editor benutzt werden.

Bei der Programmierung mit *tk-ECL^{i}PS^e* können Werkzeuge zur u.a. Fehlerdiagnose und Visualisierung von Berechnungen benutzt werden. Diese befinden sich im Menü Tools. Hier findet man u.a. den *Tracer*, mit dem Programmierer das Programm Schritt für Schritt laufen lassen und analysieren können.

5.2.3 Das Modulsystem von ECL^{i}PS^e

ECL^{i}PS^e unterstützt Objektorientierung mittels Module. Alle Bibliotheken sind Module. Ein Programm in ECL^{i}PS^e besteht aus einem oder mehreren Modulen. Es wird empfohlen unabhängige Teile eines Programms in getrennten Modulen zu halten. Dies macht Programme übersichtlich und erleichtert Programm-Modifikation.

Module werden am Anfang von Programmen mit dem Prädikat

$$:\text{-}\ module(X)$$

definiert. Dabei muß der Name des Programms und der Name des Modules, in dem das Programm geschrieben wird, übereinstimmen. In diesem Fall muss die entsprechende Programmdatei im Ordner mit X.ecl gespeichert werden.

Programme können Daten durch die Import- und Exportprädikate untereinander austauschen. Die Schnittstelle eines jeden Moduls zur Außenwelt wird durch die Objekte definiert, die aus dem Modul exportiert werden. Unterschiedliche Objekte werden unterschiedlich importiert bzw. exportiert.

Möchte man einen bestimmten Constraint-Löser X in einem Programm benutzen, so muss diese geladen werden. Dies kann entweder durch

$$:\text{-}\ lib(X) \text{ oder}$$
$$:\text{-}\ use\_modul(lib(X))$$

erfolgen.

Strukturen müssen in Modulen definiert werden, in denen sie zum Einsatz kommen. Eine Struktur Namens Buch kann wie folgt definiert werden:

$$:\text{-}\ local(buch(author, titel, verlag)).$$

Zugriff auf Daten in Strukturen ist für die Außenwelt nur möglich, wenn sie exportiert wird. Eine Struktur kann wie folgt exportiert wird:

$$:\text{- } export\ struct(buch(author, titel, verlag)).$$
$$:\text{- } local\ struct(buch(author, titel, verlag)).$$

Prädikate können nur in Modulen, in denen sie definiert sind verwendet werden. Ein Prädikat der Stelligkeit k kann wie folgt exportiert werden.

$$:\text{- } export\ p/k.$$

Kapitel 6

Simulation

6.1 Problemdarstellung

In dieser Arbeit wurde die Prüfungsordnung (Fassung von 22.07.2005) des neuen Bachelorstudiengangs Informatik der Johann Wolfgang Goethe-Universität simuliert. Die Prüfungsordnung stellt Regeln auf, die alle Objekte betreffen, die Gegenstand des Studiums sind. Die Objekte können Studenten, Professoren, Prüfungsamt-Mitarbeiter, Vorlesungen, Praktika, Prüfungen, etc. sein.

Bei der Erstellung einer Prüfungsordnung für einen neuen Studiengang müssen Entscheidungen so getroffen werden, dass sie nicht in Widerspruch mit anderen bereits getroffenen oder noch zu treffenden Entscheidungen stehen. Es können also in einer solchen Prüfungsordnung Regeln geben, die sich gegenseitig ausschließen. Solche Widersprüche könnten erst beider Anwendung der Regeln festgestellt werden, die sich in der Regel nicht so einfach beheben lassen.

Ziel der Diplomarbeit ist also, die in der Prüfungsordnung aufgestellten Regeln als Constraints zu formulieren, so dass im Idealfall logische Fehler durch eine Simulation entdeckt werden können. Weiterhin sollen Problemsituationen untersucht werden, die bei der praktischen Anwendung der Regeln auftreten könnten.

Der Bachelorstudiengang Informatik der Johann Wolfgang- Goethe-Universität besteht aus:

- den Basismodulen,
- den Vertiefungsmodulen,
- den Modulen im gewählten Anwendungsfach,
- den Ergänzungsfachmodulen und
- den Abschlussmodulen.

Diese sind voneinander getrennte Studiengebiete und werden dem entsprechend in der Prüfungsordnung unterschiedlich behandelt. Die Basismodule und das Abschlussmodul sind Pflichtmodule. Die restlichen sind Wahlpflichtmodule.

Jedes dieser Studiengebiete besteht wiederum aus Modulen, die durch Modulnummern eindeutig gekennzeichnet sind. Jedes Modul besteht aus einer oder mehreren Veranstaltungen. Die Lehrform einer Veranstaltung kann ein Praktikum, ein Seminar, eine Vorlesung, eine Tutorium-Leistung oder im Falle des Abschlussmoduls ein Oberseminar sein. Jede Veranstaltung bekommt ebenfalls eine eindeutige Veranstaltungsnummer, zugewiesen. Das Abschließen eines Moduls erfolgt durch das Bestehen der dazugehörigen Modulabschlussprüfung. Die Zulassung zu einer Modulabschlussprüfung kann das Absolvieren von Studienleistungen, die dem Modul angehören oder das Absolvieren andere Modulabschlussprüfungen voraussetzen.
Alle Module sind auf der Basis des *European Credit Transfersystems (ECTS)* mit Credit-Punkten (CP) versehen. Der Credit Punkt eines Moduls steht für den studentischen Arbeitsaufwand, der notwendig ist um den entsprechenden Anforderungen gerecht zu werden.

In den Basismodulen müssen 98 CP durch neun Modulabschlussprüfungen erzielt werden.
Die Vertiefungsmodule bestehen aus fünf Gebieten, in denen insgesamt mindestens 40 CP zu erzielen sind. In einem dieser Gebiete müssen mindestens 16 CP erreicht werden. In zwei weiteren, vom erstgenannten und untereinander verschiedenen Vertiefungsgebieten sind jeweils mindestens acht CP zu erbringen. Mit der ersten Anmeldung zu einer Modulabschlussprüfung, muss sich der Student oder die Studentin für die drei Gebiete entscheiden und diese dem Prüfungsamt mitteilen. In den 40 CP, die insgesamt zu erreichen sind, müssen mindestens ein Seminar und ein Praktikum enthalten sein. Die restlichen acht CP können beliebig erzielt werden.

Bei den Wahlpflichtmodulen ist es möglich mehr CP zu erzielen, als dies die Prüfungsordnung vorsieht. Dabei werden die CP der erfolgreichen und erfolglosen Versuche von Prüfungsleistungen mitgezählt, da die Summe insgesamt nicht mehr als 100 betragen darf. Hat der Student oder die Studentin 100 CP verbraucht, bevor die notwendigen 40 CP erreicht worden sind, so hat der Student oder die Studentin endgültig nicht bestanden.Hat der Student oder die Studentin die 40 CP erreicht, so hat er oder sie selbst zu bestimmen, welche Prüfungsleistungen mit welchen CP-Anteilen bei der Berechnung der Durchschnittsnote berücksichtigt werden sollen.

Das Abschlussmodul besteht aus dem Bachelorarbeit-Modul und dem Oberseminar-Modul. Für die Zulassung zur Bachelorarbeit müssen Studenten 100 CP bereits erzielt haben. Davon müssen mindestens 80 CP in Basismodulen erworben sein. Erzielte CP in den Anwendungsfachmodulen werden hier nicht in der Berechnung berücksichtigt.

Alle Prüfungsleistungen werden benotet, ausgenommen Prüfungsleistungen zu Praktika und zum Ergänzungsmodul.

Eine Gesamtnote für das Bachelorstudiengang errechnet sich aus dem nach CP gewichteten Durchschnitt der mit Noten bewerteten Module. Zur Gesamtnote werden 98 CP aus den Basismodulen angerechnet. Weiterhin werden, entsprechend den Erklärungen des oder der Studierenden, 40 CP aus den Vertiefungsmodulen und 24 CP aus den Anwendungsfachmodulen anzurechnen. Aufwelche Weise die 24 CP in Anwendungsfachmodulen zu erreichen sind, werden inden jeweiligen Fachrichtungen geregelt.

Bei einem Durchschnitt bis einschließlich 1,5 lautet die Gesamtnote sehr gut, bis einschließlich 2,5 gut, bis einschließlich 3,5 befriedigend, bis einschließlich 4,0 ausreichend und bei einem Durchschnitt ab 4,1 nicht ausreichend.

Bei der Berechnung der Gesamtnote wird nur die erste Dezimalstelle hinter dem Komma berücksichtigt. Alle weiteren Stellen werden ohne Rundung gestrichen.

Wir wollen im Folgenden die Problematik bei der Berechnung der Durchschnittsnote in den Vertiefungsmodulen anhand eines Beispiels erläutern.

CP-Liste:	6	8	4	9	9	8	4	9	9
Notenliste:	1,3	1,7	2,0	1,7	1,7	1,3	1,7	b	1,3

Nehmen wir an, dass der Student oder die Studentin in den Vertiefungsmodulen die oben angegebenen CP mit den entsprechenden Noten erzielt hat, wobei b für das Bestehen einer Prüfung zum Praktikum steht, das nicht benotet wird. Seien alle Module mit vier CP Seminare.
Nehmen wir weiter an, dass die ersten vier Modulabschlussprüfungen dem Gebiet entsprechen, zu dem sich der Student entschieden hat 16 CP zu erreichen. Die nächsten zwei und die letzten drei entsprechen den Gebieten, in denen jeweils acht CP zu erzielen sind. In diesem Fall hat der Student oder die Studentin 66 CP erreicht, davon werden 40 CP für die Berechnung der Durchschnittsnote berücksichtigt.

Nun soll der Student oder die Studentin bestimmen wie die 40 CP zusammengesetzt werden, so dass in den 40 CP weiterhin in allen drei Gebieten die benötigten CP erzielt worden sind, dass mindestens ein Praktikum und ein Seminar enthalten sind und natürlich dass er die bestmögliche Durchschnittsnote erzielt.

6.2 Modellierung des Problems

Bei der Modellierung kombinatorischer Probleme mit Constraint-logischen Programmiersprachen werden in der Regel die folgenden Schritte in der angegebenen Reihenfolge durchgeführt.

1. Es werden Entscheidungen getroffen, welche Datenstrukturen für die zu bearbeitenden Daten verwendet werden sollen:

Wir haben uns für die Datenstruktur Struktur entschieden, da wir sie für unsere Zwecke für geeignet halten. Strukturen mit benannten Argumenten machen die restlichen Modellierungsschritte leichter und ermöglichen schnellen Zugriff auf Daten. Ein weiterer Vorteil der Struktur ist, dass bestimmte Modifikationen unternommen werden können, ohne Programme erheblich verändern zu müssen, wie beispielsweise das Hinzunehmen neuer Argumente.

2. Problemvariablen werden generiert, die für Objekte des Problemfeldes stehen:

Die wesentlichen Gegenstände unserer Simulation sind die verschiedenen Module. Wir wollen deshalb alle Module des Bachelorstudiengangs in Strukturen speichern. Die Argumente der Strukturen entsprechen dann den verschiedenen Eigenschaften der Module. Eigenschaften sind u.a. Modulnummer, CP, Zulassungsvoraussetzungen, usw.
Wir benötigen neben den Regeln der Prüfungsordnung weitere Daten als Eingabe für das Simulationsprogramm. Das sind Informationen über den Studienverlauf eines Studenten oder einer Studentin.

Strukturen für die Module sind statisch, d.h. fest für alle Studenten gleich.Studentendaten hingegen sind individuell für jeden Studenten.

3. Constraints werden definiert:

Hier liegt die Hauptaufgabe der Simulation. Wir wollen die Regeln der Prüfungsordnung als Constraints definieren. Dabei müssen wir folgende Vereinfachungen machen:

- Eine äquivalente und formale Beschreibung von Regeln, die in einer natürlichen Sprache formuliert sind, ist bekanntlich nicht möglich. Deshalb haben wir bei der Simulation der Prüfungsordnung manche Regeln nicht mit berücksichtigt.

  Beispiel

  § 17 Abs. 3:

  „Der für den Rücktritt oder das Versäumnis einer Prüfung geltend gemachte Grund muss dem oder der Vorsitzenden des Prüfungsausschusses unverzüglich schriftlich angezeigt und glaubhaft gemacht werden. (...) ”

- Regeln, die in keinerlei Konflikte zu anderen Regeln stehen und damit für die Zielsetzung der Diplomarbeit nicht relevant sind, wurden weggelassen.

  Beispiel

  § 24 Abs. 8:

  „Auf Antrag des oder der Studierenden kann der Prüfungsausschuss die Abfassung der Bachelorarbeit in englischer Sprache zulassen, wenn das schriftliche Einverständnis des Betreuers oder der Betreuerin vorliegt."

- Aus zeitlichen Gründen wurde nur ein Teil der relevanten Regeln berücksichtigt. Dies hat zur Folge, dass die Prüfungsordnung nicht Tag-genau sondern nur auf Semester Ebene simuliert wurde.

  Beispiel

  § 16 Abs. 5:

  „(...) Die Anmeldung zu einer Klausur hat spätestens vier Wochen vor dem festgelegten Prüfungstermin und bei einer mündlichen Prüfung zu einem Prüfungszeitraum spätestens vier Wochen vor dem Beginn des Prüfungszeitraumes beim Prüfungsamt zu erfolgen.(...)."

Wie wir in den letzten Kapiteln erläutert haben, sind Constraints Bedingungen. Die Regeln der Prüfungsordnung können ebenfalls als Bedingungen aufgefaßt werden, in denen Beziehungen zwischen Objekten und deren Abhängigkeiten dargestellt werden. Wir wollen nun die formulierten Constraints zusammen mit den Informationen des Studienablaufs eines oder einer Studierenden dem ECL^{i}PS^e-System übergeben. Das System soll dann auf Anfragen über den Studienablauf des oder der Studierenden unter Beachtung der aufgestellten Constraints Antworten geben. Dies entspricht einem Constraint-Erfüllbarkeitsproblem.

Es sollen dann Fragen gestellt werden, wie z.B.:
Ist ein Student endgültig durchgefallen?, Ist ein bestimmtes Modul abgeschlossen?, Sind alle Module eines Bereichs (z.B. Basis-, Vertiefungsbereich, usw.) abgeschlossen?

Die Antwort soll nur dann ja heißen, wenn alle Constraints erfüllt sind u.a., dass bestimmte Anzahl von CP erreicht wurde , dass zu jeder Modulaschlussprüfung die jeweilige Zulassungsvoraussetzung zum Prüfungszeitpunkt bereits erreicht ist, dass ein Student oder eine Studentin nicht bereits endgültig durchgefallen ist, ...

Zusätzlich soll das System beider Berechnung der Durchschnittsnote in den Vertiefungsmodulen die bestmögliche Durchschnittsnote liefern, indem alle Kombinationen ausprobiert werden. Dies entspricht einem Constraint-Optimierungsproblem.

6.3 Implementierung

In dieser Arbeit wurde die Benutzeroberfläche *tk-ECL^{i}PS^e* zur Programmierung benutzt.

Wir haben alle Module des Bachelorstudiengangs in einem ECL^{i}PS^e-Modul gespeichert. Wir nennen dieses ECL^{i}PS^e-Modul deshalb $alle\_Module$ (siehe Anhang B1). Dieses ist eines von insgesamt acht Programmdateien, die das Simulationsprogramm ausmachen. In dieser Programmdatei werden alle Module, die zu einem Teilgebiet (Basismodule, Vertiefungsmodule, ...) gehören, in einer Liste zusammengefasst. Jede dieser Listen entspricht dem Argument eines Faktes. So werden alle Basismodule als ein einzelnes Argument des Faktes $alle\_Basismodule/1$, alle Vertiefungsmodule als ein einzelnes Argument des Faktes $alle\_Vertiefungsmodule/1$, usw.gespeichert.

Das Argument eines solchen Faktes ist also eine Liste, deren Elemente Strukturen sind. Jede einzelne Struktur in der Liste steht für einzelne Module des jeweiligen Gebiets. Die Argumente der Strukturen eines Gebiets entsprechen den verschiedenen Eigenschaften eines Moduls. Beispielsweise wurden alle Basismodulen als Strukturen wie folgt realisiert, wobei die Argumente $modul\_nr, vtg, lehrform, cp$ und $rythmus$ für die Eigenschaften des Moduls, Modulnummer, Veranstaltungsnummern, Lehrform, Credit Punkte und Häufigkeit stehen. Die Implementierung in ECL^{i}PS^e sieht wie folgt aus:

$$
\begin{array}{l}
alle\_basismodule([\\
\quad b\_modul\{modul\_nr : b\_PRG\_pr, vtg : prog\_pr, lehrform : prakt, \\
\qquad cp : 8, rhytmus : ws\}, \\
\quad b\_modul\{modul_nr : b\_HWS\_pr, vtg : hws\_pr, lehrform : prakt, \\
\qquad cp : 4, rhytmus : ss\}, \\
\quad b\_modul\{modul\_nr : b\_M1, vtg : m1, lehrform : vlg, \\
\qquad cp : 9, rhytmus : ws\}, \\
\qquad\qquad]).
\end{array}
$$

In ähnlicher Weise wurden die restlichen Module, die zu anderenTeilgebieten gehören in dem ECL^{i}PS^e-Modul $alle\_Module$ gespeichert.

Als nächstes haben wir die Daten über den Studenten oder die Studentin generiert. Diese Daten können theoretisch auf verschiede Weise dem ECL^{i}PS^e-System

übergeben werden. Beispielsweise kann man beliebige Daten aus verschieden Anwendungen heraus lesen und mit Hilfe eines Parsers in ECL^{i}PS^e-Strukturen umwandeln. Wir haben diese Daten jedoch in der Programm-Datei *student* gespeichert, die bereits in ECL^{i}PS^e-Strukturen vorliegen.

In diesem ECL^{i}PS^e-Modul werden, in getrennten Strukturen alle absolvierte Studienleistungen, alle Prüfungsversuche, die drei Wuschgebiete des oder der Studierenden gespeichert. Zu jeder dieser Leistungen des oder der Studierenden, wird vermerkt in welchem Semester die Leistung erfolgte. Diese Leistungen werden ohne Beachtung der Reihenfolge in dem Programmdatei *student* vermerkt.

Die Erstellung einer Benutzeroberfläche oder von Schnittstellen zur Dateneingabe ist eine mögliche Erweiterung des Systems, sie wurde jedoch nicht innerhalb dieser Diplomarbeit betrachet.

Bei der Formulierung der Regeln als Constraints, haben wir mit den Regeln angefangen, die sich auf die Basismodule beziehen, da diese Grundlage für weitere Teilgebiete sind und unabhängig von den restlichen Gebieten simuliert werden können. Wir haben zunächst die notwendigen Bedingungen, um ein Pflichtmodul erfolgreich abzuschließen, als benutzerdefiniertes Constraint definiert. Die Implementierung dieses Constraint in ECL^{i}PS^e erfolgt mit der Definition des folgenden Prädikats:

$ein\_Pflichtmodul\_abgesch\_basis(Modulname, Note)$:-

$das\_Modul\_ist\_ein\_Basismodul(Modul, \_CP)$,
$zulassungsvoraussetzung\_erfuellt(Modul)$,
$not\_endgueltig\_nicht\_bestanden(Modul)$,
$modulabschlusspruefung\_bestanden\_basis(Modul, Note)$.

Dieses Prädikat gelingt, wenn alle Teilziele gelingen. Dieses Prädikat kann wie folgt interpretiert werden:

Ein Pflichtmodul in den Basismodulen ist abgeschlossen genau dann, wenn

- das Modul in der Auflistung von Modulen, die in $alle\_Module$ gespeichert ist, existiert,
- der Student die Zulassungsvoraussetzungen für das Modul erfüllt,
- der Student oder die Studentin nicht bereits zweimal die Modulabschlussprüfung erfolglos wiederholt hat und
- die Modulabschlussprüfung erfolgreich absolviert hat.

Im ersten der vier Teilziele wird geprüft, ob das Modul mit der Bezeichnung *Modulname* in dem ECL^{i}PS^e-Modul *alle_Module* enthalten ist. Für das nächste Teilziel wird geprüft, ob der Student bereits die Zulassungsvoraussetzungen erfüllt hat, indem das ECL^{i}PS^e-Modul *student* durchsucht wird. Für das vierte Teilziel wird geprüft, ob ein Eintrag existiert, in dem vermerkt ist, dass der Student oder die Studentin die Modulabschlussprüfung bestanden hat.

Als nächstes haben wir alle notwendigen Bedingungen, um das Teilgebiet Basismodule erfolgreich abzuschließen, als benutzerdefiniertes Constraint realisiert. Die Implementierung in ECL^{i}PS^e sieht wie folgt aus:

$das\_Gebiet\_Basismodul\_abgesch(Liste)$:-

$finde\_nr\_des\_Moduls\_BM(Liste)$,
$pruefe\_ob\_alle\_Basismodule\_abgesch(Liste)$.

Das erste Teilziel holt sich alle Basis-Modulnummern aus der Struktur in dem ECL^{i}PS^e-Modul *basismodule* und schreibt sie in einer Liste. Das zweite Teilziel prüft, ob für jede Modulnummer X mit Hilfe von Iterationsmechanismen der Progammiersprache ECL^{i}PS^e das Prädikat $ein\_Pflichtmodul\_abgesch\_basis(X, Note)$ gelingt.

Mit der Definition eines Prädikates, das die Durchschnittsnote für Basismodulen berechnet, haben wir das dritte ECL^{i}PS^e-Modul *basismodule* vollständig erstellt.

Wir haben die weiteren ECL^{i}PS^e-Modulen, $vertiefungsmodule$, $ergaenzungsmodule$, und das *abschlussmodul* in ähnlicher Weise erstellt. Constraints die sich auf das Modul zum gewählten Anwendungsfach beziehen wurden weggelassen, da diese nicht der Informatik gehören und extern geregelt werden. Wir haben dennoch Vollständigkeitshalber dummy Prädikate in dem ECL^{i}PS^e-Modul $anwendungsfachmodule$ geschrieben.

Als besonders schwierig erwies sich zunächst die Regelung für die Berechnung der Durchschnittsnote in den Vertiefungsmodulen. Sie wurde jedoch durch eine präzise Darstellung des Problems als ein Constraint-Optimierungsproblem bewerkstelligt. Es war nicht erforderlich dem System mitzuteilen, wie die Berechnung zu erfolgen hat.

Schließlich haben wir noch ein Modul ($simulation\_bachelor$) erstellt das ein Prädikat zur Berechnung der Gesamtnote enthält. Dieses Prädikat durchläuft sämtliche Programmteile durch um das Ergebnis zu liefern, wenn in allen Gebieten die als Constraint definierten Regeln von dem oder der Studierenden erfüllt worden sind.

Im Anhang C befindet sich eine Prädikatenübersicht, wo alle Prädikate nach Gebiet aufgelistet sind.

Nun stellt sich die Frage, wer dieses Simulationsprogramm verwenden kann.
Ein Student kann das Simulationsprogramm dafür nutzen, um festzustellen, ob er alle Bedingungen für eine Modulabschlussprüfung erfüllt hat. Dies wird so gehandhabt, dass der Student oder die Studentin dem System mitteilt, dass er oder sie zuletzt die Abschlussprüfung absolviert hat. Das System prüft dann mit Hilfe des bisherigen Studienverlaufs, ob dies möglich ist.
Ist ein Constraint dabei verletzt, so antwortet das System mit *No*. Der Student oder die Studentin kann dann mit Hilfe des *tracers* eine Programmanalyse durchführen, in dem er das Programm schrittweise laufen lässt, um festzustellen welches Constraint verletzt worden ist.

Weiterhin kann das Simulationsprogramm vom dem oder der Studierenden benutzt werden, um die CP-Anteile bei der Berechnung der Durchschnittsnote zu bestimmen, so dass die bestmögliche Note erzielt werden kann. Dies ist ein kombinatorisches Problem, bei dem die Anzahl der möglichen Belegungen mit der Anzahl der erzielten CP extrem schnell wächst. Die bestmögliche Durchschnittsnote kann in Extremfällen an der Grenze zwischen zwei Bewertungen liegen, so dass der oder die Studierende eine falsche Entscheidung bezüglich der CP-Anteile treffen kann. Dies kann dazu führen, dass er oder sie beispielsweise eine Gesamtnote von gut erhält, obwohl durch andere Entscheidungen eine sehr gute Benotung möglich ist.
Auch das Prüfungsamt könnte das Simulationsprogramm in dieser Hinsicht nutzen, um Studierenden bei der Entscheidung Empfehlungen zu geben.

Wir haben in den vorherigen Kapiteln erläutert, dass bei den deklarativen Sprachen schnelle Modifikation von Programmen möglich ist, wenn Anforderungen sich ändern.
Somit kann das Simulationsprogramm mit einer Erweiterung unterschiedlichen Zwecken dienen. Beispielsweise könnte man weitere Constraints definieren, die sich u.a. aufzeitliche Verfügbarkeit von Professoren, auf vorhandene Raumkapazitäten und auf die Dauer von Vorlesungen beziehen, um Stundenpläne zu generieren.

Man könnte das Simulationsprogramm auch dahingehend erweitern, dass ein Student oder eine Studentin bei einer gegebenen Anzahl von abgeschlossenen Modulen folgende Fragen an das Simulationsprogramm stellen kann:
Welche Module können als Ergänzung absolviert werden, um das entsprechende Studiengebiet (Basis-, Vertiefungsmodulen, ...) oder um das Studium abzuschließen?

Eine weitere Erweiterung des Simulationsprogramms könnte bei der Entscheidung, welche Veranstaltung im welchem Semester (Winter- oder Sommersemester) stattfinden soll, Unterstützung leisten. Die Entscheidungen sollen so getroffen werden, dass es wie vorgegeben möglich ist,das Studium in Regelstudienzeit abzuschließen. Zusätzlich kann das Simulationsprogramm benutzt werden, um bei Änderungen oder bei Einführung von neuer Regeln festzustellen, ob die

Prüfungsordnung konsistent bleibt. Auch andere Fachrichtungen können mit leichten Erweiterungen das Simulationsprogramm für ähnliche Zwecke einsetzen.

Die Prüfungsordnung des Bachelorstudiengangs Informatik findet man auf der Webseite der Johann Wolfgang Goethe-Universität:

http://www.informatik.uni-frankfurt.de

6.4 Tests

In diesem Abschnitt wollen wir Tests führen, um zu zeigen, dass das Simulationsprogramm die gewünschten Ergebnisse liefert. Zunächst machen wir ein Test für das Constraint-Erfüllbarkeitsproblem und anschließend für das Constraint-Optimierungsproblem.

Test 1

In diesem Test wollen wir dem System folgende Informationen zu dem Studenten oder der Studentin übergeben, um anschließend die Anfrage zu stellen, ob er oder sie die Vertiefungsmodule abgeschlossen hat.
Diese Informationen sind alle absolvierten Studienleistung der Basismodule und alle Prüfungsversuche in den Basis- und Vertiefungsmodulen. Zusätzlich bekommt das System die drei Wunschgebiete des oder der Studierenden, in denen er sich verpflichtet eine bestimmte Anzahl von CP zu erzielen. Aus den Informationen soll das System erkennen, dass einige Regeln der Prüfungsordnung nicht erfüllt sind und zeigen welche Bedingung verletzt ist.

$absolvierte\_studienleistungen([$
$studienleistung\{vtg : prog1, pruefungszeit : 7, semester : ws\},$
$studienleistung\{vtg : prog2, pruefungszeit : 4, semester : ss\},$
$studienleistung\{vtg : edgi, pruefungszeit : 6, semester : ws\},$
$studienleistung\{vtg : hwr, pruefungszeit : 2, semester : ss\},$
$studienleistung\{vtg : gl2, pruefungszeit : 6, semester : ss\},$
$studienleistung\{vtg : gl1, pruefungszeit : 2, semester : ws\}$
$]).$

$pruefungsversuche\_BM([$
$pruefung\_basis\{modul\_nr : b\_HW, pruefungszeit : 4, note : 1.0\},$
$pruefung\_basis\{modul\_nr : b\_PRG, pruefungszeit : 6, note : 2.0\},$
$pruefung\_basis\{modul\_nr : b\_MOD, pruefungszeit : 1, note : 3.0\},$
$pruefung\_basis\{modul\_nr : b\_DS, pruefungszeit : 1, note : 1.0\},$
$pruefung\_basis\{modul\_nr : b\_GL, pruefungszeit : 3, note : 1.0\},$
$pruefung\_basis\{modul\_nr : b\_PRG\_pr, pruefungszeit : 6, note : b\},$
$pruefung\_basis\{modul\_nr : b\_HWS\_pr, pruefungszeit : 4, note : b\},$

$pruefung\_basis\{modul\_nr : b\_M1, pruefungszeit : 2, note : 1.0\},$
$pruefung\_basis\{modul\_nr : b\_M2, pruefungszeit : 5, note : 1.0\}$
$]).$

$pruefungsversuche\_VM([$
$pruefung\_vertiefung\{modul\_nr : b\_BS, pruefungszeit : 5, note : 1.7\},$
$pruefung\_vertiefung\{modul\_nr : b\_VS\_PR, pruefungszeit : 7, note : b\},$
$pruefung\_vertiefung\{modul\_nr : b\_BK1, pruefungszeit : 5, note : 1.7\},$
$pruefung\_vertiefung\{modul\_nr : b\_AS, pruefungszeit : 6, note : 1.7\},$
$pruefung\_vertiefung\{modul\_nr : b\_DBV, pruefungszeit : 7, note : 1.7\}$
$]).$

$drei\_Wahlgebiete\_Vertiefung([wahlgebiet\{gebiet\_1 : bkspp, gebiet\_2 : ani,$
$gebiet\_3 : gdi]).$

Auf die Anfrage
$das\_Gebiet\_Vertiefungsmodul\_abgesch(Module, CP\_Summe, Gebiete)$
antwortet das Programm mit *No*. Den Grund zeigt uns der *tracer* an, in dem er das erste nicht erfüllte Teilziel rot markiert. Von da an macht er alle vorgenommenen Schritte rückgängig und bricht das Verfahren ab. Siehe Abb. 7.1

Das System bearbeitet das folgende Prädikat für oben gestellte Anfrage:

$das\_Gebiet\_Vertiefungsmodul\_abgesch(Module, CP\_Summe, Gebiete)$:-

$suche\_alle\_bestandene\_Module\_Vertiefung/1,$
$pruefe\_ob\_alle\_Vertiefungsmodule\_regelgemaess\_abgesch/1,$
$die\_CP\_Summe\_mindestens\_40/2,$
$seminar\_und\_Praktikum\_enthalten/2,$ % **hier steckt der Fehler**
$drei\_disjunkte\_Gebiete/3.$

Test 2

Wir wollen anhand des folgenden Tests zeigen, wie schnell die Kombinationsmöglichkeit mit der Anzahl der von dem oder der Studierenden erzielten CP wächst. Dafür haben wir fünf Szenarien vorbereitet. In jeder dieser Szenarien gibt das System die bestmögliche Durchschnittsnote in den Vertiefungsmodulen an und sagt uns unter wie vielen Kandidaten (Variationen) die Lösung die beste ist.

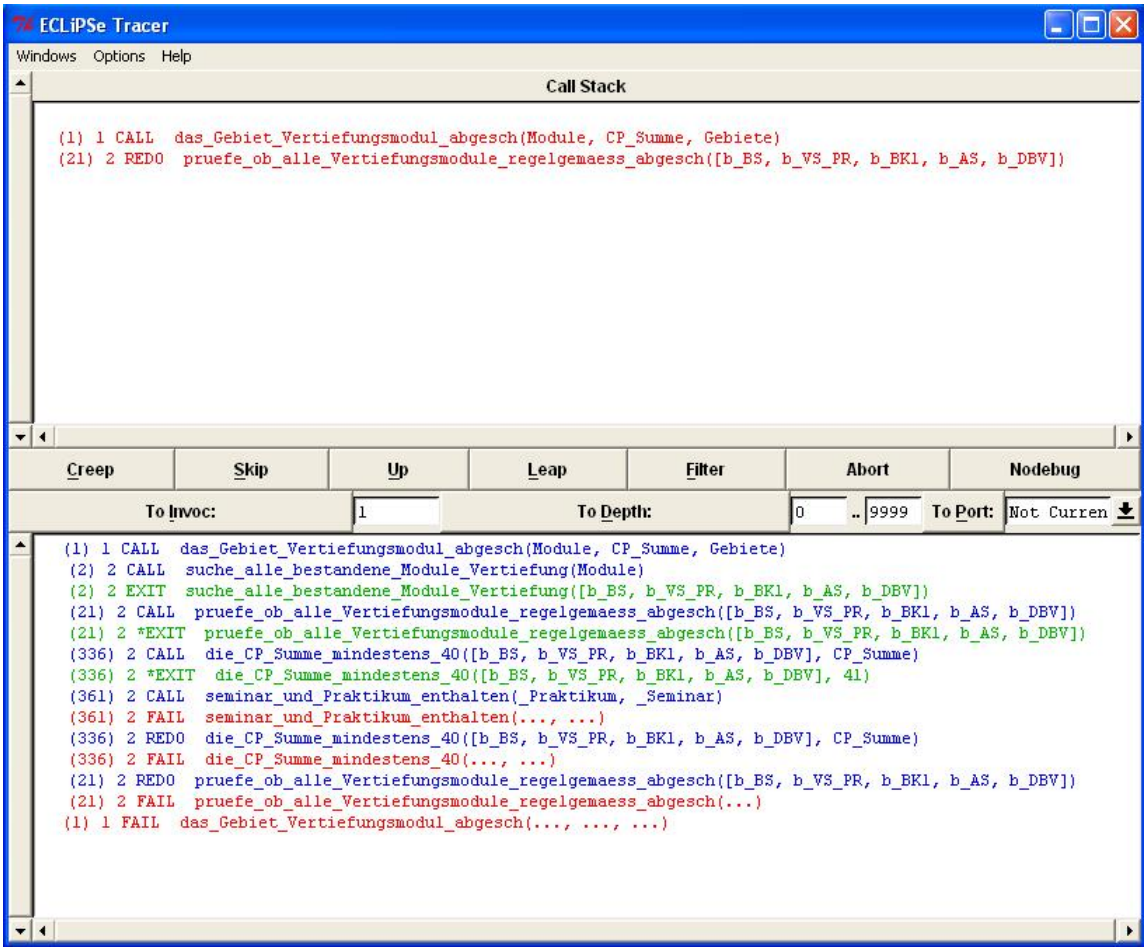

Abbildung 6.1:

Wir nehmen hier an, dass das System alle absolvierten Studienleistungen und alle Prüfungsversuche in den Basismodulen wie im obigen Test als Eingabe bekommt.

Szenario 1

$$
\begin{aligned}
&pruefungsversuche\_VM([\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BS, pruefungszeit : 5, note : 2.0\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_VS\_PR, pruefungszeit : 7, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_PR\_BS, pruefungszeit : 5, note : 1.3\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_SIM\_PR, pruefungszeit : 5, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_EAL\_BS,\\
&\qquad\qquad pruefungszeit : 1, note : 2.3\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BK1, pruefungszeit : 5, note : 1.3\}\\
&\qquad\qquad]).
\end{aligned}
$$

Insgesamt erzielte CP: 42

Szenario 2

$$
\begin{aligned}
&pruefungsversuche\_VM([\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BS, pruefungszeit : 5, note : 2.0\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_VS\_PR, pruefungszeit : 7, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_PR\_BS, pruefungszeit : 5, note : 1.3\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_SIM\_PR, pruefungszeit : 5, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_EAL\_BS,\\
&\qquad\qquad\qquad\qquad pruefungszeit : 1, note : 2.3\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BK1, pruefungszeit : 5, note : 1.3\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_VC\_PR, pruefungszeit : 6, note : b\}\\
&\qquad\qquad\qquad\qquad]).
\end{aligned}
$$

Insgesamt erzielte CP: 50

Szenario 3

$$
\begin{aligned}
&pruefungsversuche\_VM([\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BS, pruefungszeit : 5, note : 2.0\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_VS\_PR, pruefungszeit : 7, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_PR\_BS, pruefungszeit : 5, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_SIM\_PR, pruefungszeit : 5, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_EAL\_BS,\\
&\qquad\qquad\qquad\qquad pruefungszeit : 1, note : 2.3\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BK1, pruefungszeit : 5, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_VC\_PR, pruefungszeit : 6, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_AS, pruefungszeit : 6, note : 1.0\}\\
&\qquad\qquad\qquad]).
\end{aligned}
$$

Insgesamt erzielte CP: 59

Szenario 4

$$
\begin{aligned}
&pruefungsversuche\_VM([\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BS, pruefungszeit : 5, note : 2.0\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_VS\_PR, pruefungszeit : 7, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_PR\_BS, pruefungszeit : 5, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_SIM\_PR, pruefungszeit : 5, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_EAL\_BS,\\
&\qquad\qquad pruefungszeit : 1, note : 2.3\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BK1, pruefungszeit : 5, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_VC\_PR, pruefungszeit : 6, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_AS, pruefungszeit : 6, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_ASI\_PR, pruefungszeit : 5, note : b\}\\
&\qquad\qquad\qquad]).
\end{aligned}
$$

Insgesamt erzielte CP: 67

Szenario 5

$$
\begin{aligned}
&Pruefungsversuche\_VM([\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BS, pruefungszeit : 5, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_VS\_PR, pruefungszeit : 7, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_PR\_BS, pruefungszeit : 5, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_SIM\_PR, pruefungszeit : 5, note : b\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_EAL\_BS,\\
&\qquad\qquad pruefungszeit : 1, note : 2.0\}\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_BK1, pruefungszeit : 5, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_AS, pruefungszeit : 6, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_DBV, pruefungszeit : 7, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_OGL, pruefungszeit : 6, note : 1.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_KUK\_BS,\\
&\qquad\qquad pruefungszeit : 1, note : 2.7\},\\
&\quad pruefung\_vertiefung\{modul\_nr : b\_OGL, pruefungszeit : 6, note : b\}
\end{aligned}
$$

Insgesamt erzielte CP: 77

CP Gesamt	Variationen	Bearbeitungsdauer	Durchschnittsnote
42	1094	0,83 sec	1,5708333333333335
50	4833	21,71 sec	1,475
59	45816	250, 87 sec (ca. 4,18 min)	1,15
67	198625	1924,09 sec (ca. 32,07 min)	1,6999999999999995
77	Nicht genügend Speicher	Abgebrochen	-

Interpretation der Tabelle:
Die Anzahl an Variationen wächst exponentiell zur Zahl der abgeschlossenen Vertiefungsmodule. Dem entsprechend verhält sich die Laufzeit des Programms. Die Tests wurden auf einem Rechner mit 512 MB Hauptspeicher und 1,73 GHz Intel Pentium Centrino Prozessor durchgeführt. Der letzte Test mit Modulen im Umfang von 77 CP wurde aufgrund von Speichermangels abgebrochen.

Kapitel 7

Zusammenfassung

In dieser Arbeit wurde versucht die Prüfungsordnung des Bachelorstudiengangs Informatik mit Hilfe der Constraint-logischen Programmiersprache – genauer der Programmiersprache ECL^{i}PS^e – zu simulieren und diese auf logische Fehler zu überprüfen.

Hierfür wurden die beiden deklarativen Programmierparadigmen, die logische Programmierung und die Constraint-Programmierung, getrennt erläutert, da sie unterschiedliche Techniken bei der Problembehandlung einsetzen.

Zunächst wurde als Grundlage die Prädikatenlogik (Kapitel 2), erarbeitet und ausführlich dargestellt. Anschließend wurde die logische Programmierung mit der, auf Prädikatenlogik basierenden, Programmiersprache Prolog erläutert (Kapitel 3).

Nach der Erläuterung des logischen Teils wurde die Constraint-Programmierung (Kapitel 4) eingehend erläutert. Damit wurde eine Basis geschaffen, um die Constraint-logische Programmierung zu erläutern.

Die Constraint-logische Programmierung (Kapitel 5) wurde als eine Erweiterung der logischen Programmierung um Constraints und deren Behandlung dargestellt. Dabei wurde zunächst ein allgemeiner Ansatz der Constraint-logischen Programmierung (CLP-Paradigma) erläutert. Mit der Einführung der Programmiersprache ECL^{i}PS^e, wurden alle Werkzeuge behandelt, die für die Simulation benötigt wurden.

Schließlich wurde genauer auf die Modellierung des Problems und seine Implementierung in der Sprache ECL^{i}PS^e eingegangen (Kapitel 6). Grundidee der Simulation war, die Regeln der Prüfungsordnung als Constraints zu formulieren, so dass sie formal bearbeitet werden konnten. Hier wurden zwei Arten von Tests durchgeführt:

- Constraint-Erfüllbarkeitsproblem

  Mit dem ersten Test wurde nach eine Lösung gesucht, in der alle Constraints erfüllt sind.

- Constraint-Optimierungsproblem

  Hier wurde nach einer optimalen Lösung gesucht unter mehreren Kandidaten, in der alle Constraints erfüllt sind.

Fazit

Die Constraint-logische Programmierung ist ein viel versprechendes Gebiet, da sie ein Mittel zur Behandelung kombinatorischer Probleme darstellt. Solche Probleme treten in vielen verschiedenen Berufsfeldern auf und lassen sich sonst nur mit großem Aufwand bewältigen. Beim Auftreten solcher Probleme kann schnell ein konzeptuelles Modell erstellt werden, das sehr einfach in ein ausführbares Programm (Design-Modell) umgewandelt wird. Programm-Modifikation ist erheblich leichter als in den prozeduralen Programmiersprachen.

Anhang A

Simulationsprogramm

A.1 Das Modul: alle_Module

:- $module(alle\_Module)$.

% Der Name des Modules wird definiert.

:- $lib(ic)$.
:- $exportalle\_basis\_Module/1$.
:- $exportalle\_Vertiefungsmodule/1$.
:- $exportalle\_Anwendungsfachmodule/1$.
:- $exportalle\_Ergaenzungsmodule/1$.
:- $exportdas\_Abschlussmodul/1$.

% Hier werden Strukturen exportiert.

:- $exportstruct(b\_modul(modul\_nr, vtg, lehrform, cp, rhytmus))$.
:- $exportstruct(v\_modul(modul\_nr, gebiet, lehrform, cp, rhytmus, vorauss))$.
:- $exportstruct(a\_modul(modul\_nr, cp, rhytmus))$.
:- $exportstruct(erg\_modul(modul\_nr, cp, lehrform, rhytmus, vorauss))$.
:- $exportstruct(afm\_modul(modul\_nr, cp, durchschnittsnote))$.

% Hier werden Struktur Definitionen exportiert.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1.0

/* Hier wird dem System alle wichtige Informationen über Basismodulen mitgeteilt. Die Struktur $alle\_basis\_Module$ enthält alle Module desVertiefungsgebietes. */

$alle\_basis\_Module$([

$b\_modul\{modul\_nr : b\_PRG, vtg : prog1, lehrform : vlg, cp : 17, rhytmus : ws\}$,
$b\_modul\{modul\_nr : b\_PRG, vtg : prog2, lehrform : vlg, cp : 17, rhytmus : ss\}$,

$b\_modul\{modul\_nr : b\_HW, vtg : edgi, lehrform : vlg, cp : 14, rhytmus : ws\},$
$b\_modul\{modul\_nr : b\_HW, vtg : hwr, lehrform : vlg, cp : 14, rhytmus : ss\},$

$b\_modul\{modul\_nr : b\_MOD, vtg : mod, lehrform : vlg, cp : 7, rhytmus : ws\},$

$b\_modul\{modul\_nr : b\_DS, vtg : ds, lehrform : vlg, cp : 5, rhytmus : ss\},$

$b\_modul\{modul\_nr : b\_GL, vtg : gl1, lehrform : vlg, cp : 16, rhytmus : ws\},$
$b\_modul\{modul\_nr : b\_GL, vtg : gl2, lehrform : vlg, cp : 16, rhytmus : ss\},$
$b\_modul\{modul\_nr : b\_PRG\_pr, vtg : prog\_pr, lehrform : prakt,$
$cp : 8, rhytmus : ws\},$

$b\_modul\{modul\_nr : b\_HWS\_pr, vtg : hws\_pr, lehrform : prakt,$
$cp : 4, rhytmus : ss\},$

$b\_modul\{modul\_nr : b\_M1, vtg : m1, lehrform : vlg, cp : 9, rhytmus : ws\},$

$b\_modul\{modul\_nr : b\_M2, vtg : m2\_a, lehrform : vlg, cp : 18, rhytmus : ws\},$
$b\_modul\{modul\_nr : b\_M2, vtg : m2\_b, lehrform : vlg, cp : 18, rhytmus : ss\},$
$b\_modul\{modul\_nr : b\_M2, vtg : m2\_c, lehrform : vlg, cp : 18, rhytmus : ss\}]).$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2.0

/* Hier wird demSystem alle wichtige Informationen über Vertiefungsmodulen mitgeteilt. Die Struktur $alle\_Vertiefungsmodule$ enthält alle Moduledes Vertiefungsgebietes. */

$alle\_Vertiefungsmodule([$
$v\_modul\{modul\_nr : b\_VS, gebiet : bkspp, lehrform : vlg, cp : 6,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_VS\_PR, gebiet : bkspp, lehrform : prkt, cp : 8,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_BS, gebiet : bkspp, lehrform : vlg, cp : 9,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_ST, gebiet : bkspp, lehrform : vlg, cp : 6, r$
$hytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_EPF, gebiet : bkspp, lehrform : vlg, cp : 5,$
$rhytmus : ws, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_PR\_BS, gebiet : bkspp, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_KS\_BS, gebiet : bkspp, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\},$

%% Ende Vertiefungsgebiet (bkspp) Betriebs- und Kommunikationssysteme
%% und Programmiersprachen und -Paradigmen

$v\_modul\{modul\_nr : b\_DB1, gebiet : iswv, lehrform : vlg, cp : 9,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_DB2, gebiet : iswv, lehrform : vlg, cp : 6,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_IS\_BS, gebiet : iswv, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_KI, gebiet : iswv, lehrform : vlg, cp : 6,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_WV\_BS, gebiet : iswv, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_CLT, gebiet : iswv, lehrform : vlg, cp : 9,$
$rhytmus : zj, vorauss : 2\},$
$v\_modul\{modul\_nr : b\_AS, gebiet : iswv, lehrform : vlg, cp : 9,$
$rhytmus : zj, vorauss : 3\},$
$v\_modul\{modul\_nr : b\_AS\_PR, gebiet : iswv, lehrform : prkt, cp : 8,$
$rhytmus : zj, vorauss : 3\},$
$v\_modul\{modul\_nr : b\_AS\_BS, gebiet : iswv, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\},$

%% Ende Vertiefungsgebiet (iswv) Informationssysteme und Wissensverarbeitung

$v\_modul\{modul\_nr : b\_RA, gebiet : ts, lehrform : vlg, cp : 6,$
$rhytmus : ws, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_RT, gebiet : ts, lehrform : vlg, cp : 6,$
$rhytmus : ws, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_ES, gebiet : ts, lehrform : vlg, cp : 6,$
$rhytmus : zj, vorauss : 4\},$
$v\_modul\{modul\_nr : b\_AIS, gebiet : ts, lehrform : vlg, cp : 6,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_REM, gebiet : ts, lehrform : vlg, cp : 6,$
$rhytmus : ss, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_EM\_BS, gebiet : ts, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_SYSA\_BS, gebiet : ts, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_ASI\_PR, gebiet : ts, lehrform : prkt, cp : 8,$
$rhytmus : zj, vorauss : 1\},$

%% Ende Vertiefungsgebiet (ts) Technische Systeme

$v\_modul\{modul\_nr : b\_SIM, gebiet : ani, lehrform : vlg, cp : 6,$
$rhytmus : zj, vorauss : 3\},$
$v\_modul\{modul\_nr : b\_SIM\_PR, gebiet : ani, lehrform : prkt, cp : 8,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_CG, gebiet : ani, lehrform : vlg, cp : 6,$
$rhytmus : ss, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_HCI, gebiet : ani, lehrform : vlg, cp : 4,$
$rhytmus : ws, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_OGL, gebiet : ani, lehrform : vlg, cp : 6,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_MMS, gebiet : ani, lehrform : vlg, cp : 3,$
$rhytmus : ss, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_STCG, gebiet : ani, lehrform : vlg, cp : 3,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_VC\_PR, gebiet : ani, lehrform : prkt, cp : 8,$
$rhytmus : ws, vorauss : 5\},$
$v\_modul\{modul\_nr : b\_DBV, gebiet : ani, lehrform : vlg, cp : 6,$
$rhytmus : ss, vorauss : 6\},$
$v\_modul\{modul\_nr : b\_ANI\_BS, gebiet : ani, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\},$

%% Ende Vertiefungsgebiet Angewandte Informatik

$v\_modul\{modul\_nr : b\_EAL, gebiet : gdi, lehrform : vlg, cp : 9,$
$rhytmus : ss, vorauss : 2\},$
$v\_modul\{modul\_nr : b\_KRY, gebiet : gdi, lehrform : vlg, cp : 9,$
$rhytmus : zj, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_MFS\_bs, gebiet : gdi, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 7\},$
$v\_modul\{modul\_nr : b\_BK1, gebiet : gdi, lehrform : vlg, cp : 9,$
$rhytmus : zj, vorauss : 7\},$
$v\_modul\{modul\_nr : b\_KUK\_BS, gebiet : gdi, lehrform : smn, cp : 4,$
$rhytmus : js, vorauss : 1\},$
$v\_modul\{modul\_nr : b\_EAL\_BS, gebiet : gdi, lehrform : smn, cp : 4,$
$rhytmus : zj, vorauss : 1\}$
$]).$

%% Ende Vertiefungsgebiet Grundlage der Informatik

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3.0

/* Hier wird demSystem alle wichtige Informationen über die Module zum gewählten Anwendungsfach mitgeteilt. Die Struktur $alle\_Anwendungsfachmodule$ enthält alle_module des Vertiefungsgebietes. */

$$
\begin{array}{l}
alle\_Anwendungsfachmodule([\\
\quad afm\_modul\{modul\_nr: x, cp: 24, durchschnittsnote: 3.7\}\\
\qquad\qquad\qquad]).
\end{array}
$$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 4.0

/* Hier wird demSystem alle wichtige Informationen über Ergänzungsmodule mitgeteilt. Die Struktur $alle\_Ergaenzungsmodule$ enthält alle Module des Vertiefungsgebietes. */

$$
\begin{array}{l}
alle\_Ergnzungsmodule([\\
\quad erg\_modul\{modul\_nr: b\_TL, cp: 3, lehrform: tut\_ltg,\\
\qquad\qquad rhytmus: js, vorauss: 1\},\\
\quad erg\_modul\{modul\_nr: b\_NMG, cp: 3, lehrform: vlg,\\
\qquad\qquad rhytmus: ws, vorauss: 1\},\\
\quad erg\_modul\{modul\_nr: b\_PM, cp: 3, lehrform: vlg,\\
\qquad\qquad rhytmus: ws, vorauss: 1\}\\
\qquad\qquad\qquad]).
\end{array}
$$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5.0

/* Hier wird dem Systemalle wichtige Informationen über das Abschlussmodul mitgeteilt. Die Struktur $das\_Abschlussmodul$ enthält alle Module desVertiefungsgebietes. */

$$
\begin{array}{l}
das\_Abschlussmodul([\\
\quad a\_modul\{modul\_nr: b\_BA\_os, cp: 3, rhytmus: js\},\\
\quad a\_modul\{modul\_nr: b\_BA\_ba, cp: 12, rhytmus: js\}\\
\qquad\qquad\qquad]).
\end{array}
$$

A.2 Das Modul: basismodule

:- $module(basismodule)$.

% Der Name des Modules wird definiert.

:- $export(schreibe\_alle\_erzielte\_CP\_basis/1)$.
:- $export(ein\_Pflichtmodul\_basis\_abgesch/2)$.
:- $export(berechne\_Durchschnittsnote\_Basismodule/3)$.
:- $use\_module(student)$.
:- $use\_module(alle\_Module)$.

% Hier werden Module geladen, die in diesem Modul aufgerufen werden.

:- $lib(ic)$.

% Die ic-Bibliothek wird geladen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prädikat 1.

/*Mit diesem Prädikat werden alle notwendige Bedingungen für das Abschließen eines Basismoduls definiert. */

$ein\_Pflichtmodul\_basis\_abgesch(Modul, Note)$:-
$das\_Modul\_ist\_ein\_Basismodul(Modul, \_CP\_Zahl)$,

% Hier wird geprüft, ob Modul ein der Datenbank bekannte Basismodule ist.

$zulassungsvoraussetzung\_erfuellt(Modul)$,

% Hier wird geprüft, ob Zulassungsvoraussetzungen für die Modulabschlussprüfung erfüllt sind.

$not\_endgueltig\_nb(Modul)$,

% Der Student ist noch nicht in der Modulabschlussprüfung drei mal durchgefallen bzw. endgültig nicht bestanden.

$modulabschlusspruefung\_bestanden\_basis(Modul, Note)$.

/* Im folgenden werden Klausel definiert, die in Prädikat 1 benutzt wurden.*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%1.1

$das\_Modul\_ist\_ein\_Basismodul(Modul, CP\_Zahl)$:-
$alle\_basis\_Module(W)$,
$member(b\_modul\{modul\_nr : Modul, cp : CP\_Zahl\}, W)$.

% Hier wird geprüft, ob Modul ein Basismodul ist und wenn ja, die CP des Modules wird der Variable CP_Zahl zugeordnet.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1.2

$zulassungsvoraussetzung\_erfuellt(Modul)$:-
$(Modul = b\_PRG- >$
$einer\_der\_Studienleistungen\_vorher\_erbracht(b\_PRG, prog1, prog2)$
$; true),$
$(Modul = b\_PROG\_pr- >$
$einer\_der\_Studienleistungen\_vorher\_erbracht(b\_PROG\_pr, prog1, prog2)$
$; true),$
$(Modul = b\_HW- >$
$einer\_der\_Studienleistungen\_vorher\_erbracht(b\_HW, edgi, hwr)$
$; true),$
$(Modul = b\_GL- >$
$einer\_der\_Studienleistungen\_vorher\_erbracht(b\_GL, gl1, gl2)$
$; true),$
$(Modul = b\_HWS\_pr- >$
$studienleistung\_vorher\_erbracht\_HWS(b\_HWS\_pr)$
$; true),$
$(Modul = b\_M2- > studienleistung\_vorher\_erbracht\_M2(b\_M2)$
$; true).$

% Hier wird sechs mal geprüft, ob die Variable Modul eine bestimmte Modulnummer ist und je nach Nummer werden Klauseln aufgerufen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1.2.1

/* Vorausetzung für die Modulabschlussprüfungen b_PRG, b_PRG_pr, b_HW und b_GL */

$einer\_der\_Studienleistungen\_vorher\_erbracht(Modulname, Studleist1, \_)$:-

$berechne\_pruefungszeitpunkt(Modulname, Semester\_Zahl1)$,
$absolvierte\_studienleistungen(W)$,
$member(studienleistung\{vtg : Studleist1, pruefungszeit : Zeitpunkt\}, W)$,
$Zeitpunkt \#=< Semester\_Zahl$,

```
einer_der_Studienleistungen_vorher_erbracht(Modulname, _, Studleist2):-
    berechne_pruefungszeitpunkt(Modulname, Semester_Zahl),
    absolvierte_studienleistungen(W),
    member(studienleistung{vtg : Studleist2, pruefungszeit : Zeitpunkt}, W),
    Zeitpunkt #=< Semester_Zahl.

% Hier wird geprüft, ob eine der Sudienleistungen bereits absolviert ist.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1.2.2

/* Vorausetzung für die Modulabschlusspr. b_HWS_pr */

studienleistung_vorher_erbracht_HWS(b_HWS_pr) :-
    berechne_pruefungszeitpunkt(b_HWS_pr, Semester_Zahl),
    pruefungsversuche_BM(W),
    member(pruefung_basis{modul_nr : b_HW,
                pruefungszeit : Zeitpunkt, note : X}, W),
    not(X = 5.0),
    Zeitpunkt# =< Semester_Zahl, !.


studienleistung_vorher_erbracht_HWS(b_HWS_pr) :-
    berechne_pruefungszeitpunkt(b_HWS_pr, Semester_Zahl),
    pruefungsversuche_BM(W),
    member(pruefung_basis{modul_nr : b_PRG,
                pruefungszeit : Zeitpunkt, note : X}, W),
    not(X = 5.0),
    Zeitpunkt# =< Semester_Zahl, !.


studienleistung_vorher_erbracht_HWS(b_HWS_pr):-
    berechne_pruefungszeitpunkt(b_HWS_pr, Semester_Zahl),
    pruefungsversuche_BM(W),
    member(pruefung_basis{modul_nr : b_MOD,
                pruefungszeit : Zeitpunkt1, note : X1}, W),
    member(pruefung_basis{modul_nr : b_DS,
                Pruefungszeit : Zeitpunkt2, note : X2}, W),
    not(X1 = 5.0),
    not(X2 = 5.0),
    Zeitpunkt1 #=< Semester_Zahl,
    Zeitpunkt2 #=< Semester_Zahl.
```

% Hier wird geprüft, ob das Modul b_HW oder das Modul b_PRG oder die beiden Module b_MOD und b_DS abgeschlossen sind.

%%%
%% 1.2.3

/* Vorausetzung für die Modulabschlusspr.b_M2 */

```
studienleistung_vorher_erbracht_M2(b_M2) :-
    berechne_pruefungszeitpunkt(b_M2, Semester_Zahl),
    pruefungsversuche_BM(W),
    member(pruefung_basis{modul_nr : b_M1,
                pruefungszeit : Zeitpunkt, note : X}, W),
    not(X = 5.0),
    Zeitpunkt #=< Semester_Zahl.
```

% Hier wird geprüft, ob das Modul b_M1 abgeschlossen ist.

%%%
%% 1.2.3.1

```
berechne_pruefungszeitpunkt(Modulname, Semester_Zahl) :-
    pruefungsversuche_BM(W),
    member(pruefung_basis{modul_nr : Modulname,
                pruefungszeit : Y}, W),
    Semester_Zahl is Y.
```

% Mit dieser Klausel wird die Zeit, in der die Modulabschlussprüfung gemacht worden ist, der Variable *Semester_Zahl* zugeordnet.

%%%
%% 1.3

```
not_endgueltig_nb(Modul) :-
    pruefungsversuche_BM(W1),
    (foreach(Z1, W1), fromto(W2, Out, In, []), param(Modul) do
    arg(1, Z1, Modul)- > Out = [Z1|In]
    ;
    Out = In),
```

% Hier werden alle Prüfungsversuche im selben Modul rausgefiltert und in die Liste W2 geschrieben.

```
    (foreach(Z2, W2), fromto(W3, Out, In, []) do
    arg(3, Z2, 5.0)- > Out = [Z2|In]; Out = In),
    length(W3, D),
    D# < 3.
```

% Hier werden alle fehlgeschlagene Prüfungsversuche rausgefiltert und in die Liste $W3$ geschrieben, mit der Bedingung dass, $W3$ weniger als 3 Elemente enthalten darf.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1.4

$modulabschlusspruefung\_bestanden\_basis(Modul, N)$:-
$pruefungsversuche\_BM(W)$,
$member(pruefung\_basis\{modul\_nr : Modul, note : N\}, W)$,
$not(N = 5.0)$.

% Mit dieser Klausel wird geprüft, ob der Student die Modulabschlussprüfung bestanden hat und wenn ja, die Note wird der Variable N zugeordnet.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prädikat 2.

$das\_Gebiet\_Basismodul\_abgesch(Liste)$:-
$finde\_alle\_Modul\_nr\_BM(Liste)$,

% Hier werden alle Modulnummern der Basismodule der Variable Liste zugeordnet.

$pruefe\_ob\_alle\_Basismodule\_abgesch(Liste)$.

% Hier wird für jedes Modul geprüft, ob Student das Modul abgeschlossen hat.

/* Im folgenden werden Klauseln defniert, die im Prädikat 2. benutzt wurden.*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2.1

$finde\_alle\_Modul\_nr\_BM(Liste)$:-
$alle\_basis\_Module(Y)$,

% Hier wird der Variable Y die Struktur zugeordnet, die alle Basismodule enthält.

$(foreach(X1, Y), foreach(P1, Temp)$
$do\ finde\_nr\_des\_Moduls\_BM(X1, P1))$,

% Hier wird iterativ die Modulnummer eines jeden Basismoduls in der Liste Temp geschrieben.

$sort(Temp, Liste)$.

% Hier werden verdopplungen eliminiert.

% Mit dieser Klausel werden alle Modulnummer der Basismodule gesammelt, und der Variable Liste zugeordnet.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2.1.1

$finde\_nr\_des\_Moduls\_BM(b\_modul\{modul\_nr : X\}, Z)$:-
$Z = X$.

/* Mit dieser Klausel wird der Variable Z die Modulnummer (X) zugeordnet. */

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2.2

$pruefe\_ob\_alle\_Basismodule\_abgesch(Liste)$:-
$(foreach(X, Liste)\ do$
$ein\_Pflichtmodul\_basis\_abgesch(X, \_Note))$.

% Hier wird für jede Modulnummer in der Liste iterativ geprüft, ob der Student die einzelnen Module abgeschlossen hat.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prädikat 3.

$berechne\_Durchschnittsnote\_Basismodule(Durchschnittsnote,$
$CP\_Total, CP\_benot)$:-
$das\_Gebiet\_Basismodul\_abgesch(\_Liste),$
$schreibe\_alle\_erzielte\_Noten\_basis(T1),$

% In $T1$ werden alle Noten geschrieben.

$schreibe\_alle\_erzielte\_CP\_basis(T2),$

% In $T2$ werden alle CP geschrieben.

$pruefe\_ob\_CP\_Summe\_min\_98(T2, CP\_Total),$

% Hier wird geprüft, ob 98 CP in Basismodulen erzielt worden sind.

$gewichte\_numerische\_Noten\_durch\_CP\_1(T1, T2, Gesamt),$

% Hier werden numerische Noten durch CP gewichtet und die Summe wird der Variable *Gesamt* zugeordnet.

$bilde\_CP\_Summe\_der\_numerischen\_Noten(T1, T2, CP\_benot),$

% Hier werden die CP der numerischen Noten addiert und der der Variable CP_benot zugeordnet.

$Durchschnittsnote\ is\ Gesamt/CP\_benot,$

% Mit dem eingebauten Prädikat is / 3 wird der Durchschnitt berechnet.

$print('Durchschnittsnote :'),$
$print(Durchschnittsnote), nl.$

% Hier wird das Ergebnis ausgegeben.

/* Im folgenden werden Klauseln definiert, die im Prädikat 3. benutzt wurden.*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3.1

$schreibe\_alle\_erzielte\_Noten\_basis(T)$:-
$finde\_alle\_Modul\_nr\_BM(Liste),$
$(foreach(X, Liste), foreach(P, T) do$
$modulabschlusspruefung\_bestanden\_basis(X, P)).$

% Alle benoteten und unbenoteten Noten werden in die Liste T geschrieben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3.2

$schreibe\_alle\_erzielte\_CP\_basis(T)$:-
$finde\_alle\_Modul\_nr\_BM(Liste),$
$(foreach(X, Liste),$
$foreach(P, T) do$
$das\_Modul\_ist\_ein\_Basismodul(X, P)).$

% Alle erzielten CP werden in die Liste T geschrieben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3.3

$pruefe\_ob\_CP\_Summe\_min\_98(T2, CP\_Total)$:-
$sum(T2, CP\_Total),$
$CP\_Total\# >= 98.$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3.4

```
gewichte_numerische_Noten_durch_CP_1(T1, T2, Gesamt):-
    (foreach(X, T1),
    foreach(Y, T2),
    foreach(Z, T3)do
    real(X)-> *(X, Y, Z)
    ;
    Z = 0),
    sum(T3, Gesamt).
```

% Alle numerischen Noten werden mit gewichtet und die Summe wird der Variable Gesamt zugeordnet.

%%
%% 3.5

```
bilde_CP_Summe_der_numerischen_Noten(T1, T2, CP_benot) :-
    (foreach(X, T1),
    foreach(Y, T2),
    foreach(Z, T4)do
    real(X)-> Z = Y
    ;
    Z = 0),
    sum(T4, CP_benot).
```

% Nicht numerische Noten bekommen den CP-Wert 0 und die restlichen werdenzusammenaddiert.

A.3 Das Modul: anwendungsfachmodule

:- $module(anwendungsfachmodule)$.

% Der Name des Modules wird definiert.

:- $export(anwendungsfachmodule\_abgeschlossen/2)$.
:- $use\_module(student)$.
:- $use\_module(alle\_Module)$.

% Hier werden Module geladen, die in diesem Modul aufgerufen werden.

:- $lib(ic)$.

% Die ic-Bibliothek wird geladen.

$anwendungsfachmodule\_abgeschlossen(Durchschnittsnote, CP\_Anwendungsfach)$:-
$alle\_Anwendungsfachmodule(W)$,
$member(afm\_modul\{cp : CP\_Anwendungsfach, durchschnittsnote : Durchschnittsnote\}, W)$.

A.4 Das Modul: ergaenzungsmodule

:- $module(ergaenzungsmodule)$.

% Der Name des Modules wird definiert.

:- $export(erzielte\_CP\_Ergaenzung/1)$.

% Hier wird das Prädikat ausserhalb des Modules Sichtbargemacht.

:- $use\_module(student)$.
:- $use\_module(alle\_Module)$.
:- $use\_module(basismodule)$.

%Hier werden Module geladen, die in diesem Modul aufgerufen werden.

:- $lib(ic)$.

% Hier wird die IC-Bibliothek geladen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prädikat 8.

$ergaenzungsmodul\_abgeschlossen(Modul, CP\_Total)$:-
$erfolgreiche\_pruefungsversuche\_EM(W)$,
$member(ergaenzungsmodul\{modul\_nr : Modul, pruefungszeit : Z\}, W)$,
$voraussetzung\_Ergaenzungsmodul(Z)$,
$voraussetzung\_wenn\_X\_tutorium(Modul, Z)$,
$erzielte\_CP\_Ergaenzung(CP\_Total)$.

% Hier werden alle Voraussetzungen definiert, die für das Abschließen der Ergänzungsmodule notwendig sind.

/* Im Folgenden werden 8 Klauseln definiert,die im Prädikat 8 benutzt wurden.*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 8.1

$voraussetzung\_Ergaenzungsmodul(Zeit)$:-
$pruefungsversuche\_BM(W)$,
$member(pruefung\_basis\{modul\_nr : b\_HW,$
$pruefungszeit : Zeitpunkt\}, W)$,
$Zeitpunkt \#=< Zeit$, !.

% Hier wird geprüft, ob das Modul b_HW bereitsabgeschlossen ist.

```
voraussetzung_ergaenzung(Zeit) :-
    pruefungsversuche_BM(W),
    member(pruefung_basis{modul_nr : b_PRG,
            pruefungszeit : Zeitpunkt}, W),
    Zeitpunkt4#=< Zeit, !.
```

% Hier wird geprüft, ob das Modul b_PRG bereits abgeschlossen ist.

```
voraussetzung_ergaenzung(Zeit) :-
    pruefungsversuche_BM(W),
    member(pruefung_basis{modul_nr : b_MOD,
    pruefungszeit : Zeitpunkt1}, W),
    member(pruefung_basis{modul_nr : b_DS,
            pruefungszeit : Zeitpunkt2}, W),
    Zeitpunkt1#=< Zeit,
    Zeitpunkt2#=< Zeit.
```

% Hier wird geprüft, ob beide Module (b_MOD und b_DS) bereits abgeschlossen sind.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % 8.2

```
voraussetzung_wenn_X_tutorium(X, Z) :-
    (X = [b_TL, MOD]- >
    bereits_abgesch(MOD, Z)
    ;
    member(X, [b_NMG, b_PM])
    ).
```

% Hier geprüft, ob der Student eine Tutoriumleistung im Ergänzungsmodul abgeschlossen hat. Wenn ja, muss der Student das entsprechende Modul abgeschlossen haben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 8.2.1

```
bereits_abgesch(Modul, Z1) :-
    pruefungsversuche_BM(W),
    member(pruefung_basis{modul_nr : Modul,
            pruefungszeit : Z2, note : N}, W),
    ein_Pflichtmodul_basis_abgesch(Modul, _Note),
    N =< 4.0,
    Z2 #=< Z1, !.

bereits_abgesch(Modul, Z1) :-
```

```
    pruefungsversuche_VM(W),
    member(pruefung_vertiefung{modul_nr : Modul,
            pruefungszeit : Z2, note : N}, W),
    N =< 4.0,
    Z2 #=< Z1.
```

% Hier wird geprüft, ob der Student das entsprechende Modul im Basis- oderVertiefungsmodule abgeschlossen hat.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 8.3

erzielte_CP_Ergaenzung(Total) :-
    erfolgreiche_pruefungsversuche_EM(W),
    (foreach(X, W),
    foreach(Y, L) do
    berechne_CP_Total(X, Y)
    ),
    sum(L, Total).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 8.3.1

berechne_CP_Total(ergaenzungsmodul{modul_nr : P, pruefungszeit : 5}, Y) :-
    alle_Ergaenzungsmodule(R),
    (P = [b_TL|_]- >
    member(erg_modul{modul_nr : b_TL, cp : Y}, R)
    ;
    (P = Modul),
    member(erg_modul{modul_nr : Modul, cp : Y}, R)
    ).
```

% Hier wird das erreichte CP gerechnet und geprüft, für den Fall dass der Student Tutoriumsleistung erbracht hat, ob er das entsprechende Modul abgeschlossen hat.

A.5 Das Modul: vertiefungsmodule

:- $module(vertiefungsmodule)$.

% Der Name des Modules wird definiert.

:- $export(pruefe\_ob\_alle\_Vertiefungsmodule\_regelgemaess\_abgesch/1)$.
:- $export(suche\_alle\_bestandene\_Module\_Vertiefung/1)$.
:- $export(finde\_CP\_des\_Moduls/2)$.
:- $export(ein\_Wahlpflichtmodul\_abgesch\_Vertiefung/1)$.
:- $export(finde\_best\_moegliche\_durchschnittsnote\_Vertiefung/2)$.
:- $export(berechne\_Durchschnittsnote\_Vertiefung/5)$.

:- $use\_module(student)$.
:- $use\_module(alle\_Module)$.
:- $use\_module(basismodule)$.

% Hier Werden Module geladen, die in diesem Modul aufgerufen werden.

:- $lib(ic)$.

% Die ic-Bibliothek wird geladen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prädikat 4.

$ein\_Wahlpflichtmodul\_abgesch\_Vertiefung(Modul)$:-

$das\_Modul\_ist\_ein\_Vertiefungsmodul(Modul, Voraussetzung)$,

% Modul ist ein Vertiefungsmodul.

$nicht\_bereits\_2\_mal\_Pruefung\_wiederholt(Modul)$,

% Der Student hat nicht bereits zweimal dieModulabschlussprüfung wiederholt.

$cp\_Konto\_nicht\_ueberlastet(\_Modul)$,

% Hier wird geprüft, ob der Student noch freie Versuche hat.

$modulabschlusspruefung\_bestanden\_Vertiefung(Modul, Zeit)$,

% Student hat das Modul mit Erfolg abgeschlossen.

$zulassungsvoraussetzung\_Vertiefung(Voraussetzung, Zeit)$.

% Zulassungsvoraussetzung erfüllt.

/* Im folgenden werden Klauselndefiniert, die in Prädikat 4. benutztwurden.*/

%%
%% 4.1

$das\_Modul\_ist\_ein\_Vertiefungsmodul(M, V)$:-

$alle\_Vertiefungsmodule(Y)$,
$member(v\_modul\{modul\_nr : M, vorauss : V\}, Y)$.

%Hier wird geprüft, ob M ein gültige Bezeichnung eines Vertiefungsmoduls istund wenn ja, wird der Variable V die für das Modul geltende Vorausetzung ($vorauss$)zugeordnet.

%%
%% 4.2

$nicht\_bereits\_2\_mal\_Pruefung\_wiederholt(M)$:-

$pruefungsversuche\_VM(W1)$,
$(foreach(Z1, W1), fromto(W2, Out, In, []), param(M)\ do$
$Z1 = [M|\_]- > Out = [Z1|In]$
$; Out = In)$,

%Hier werden alle Prüfungsversuche im selben Modul rausgefiltert und in dieListe W2 geschrieben.

$(foreach(Z2, W2), fromto(W3, Out, In, [])\ do$
$arg(4, Z2, 5.0)- > Out = [Z2|In]; Out = In)$,
$length(W3, D)$,
$D \#< 3$.

% Hier werden alle fehlgeschlagene Prüfungsversuche rausgefiltert und in die Liste W3 geschrieben, mit der Bedingung dass, W3 weniger als 3 Elemente enthalten darf.

%%
%% 4.3

$cp\_Konto\_nicht\_ueberlastet(C)$:-

$pruefungsversuche\_VM(W)$,
$(foreach(X, W)$,
$foreach(Y, T1)\ do$
$finde\_die\_Modul\_nr(X, Y))$,
$(foreach(X, T1)$,

$foreach(Y, T2)\ do$
$finde\_CP\_des\_Moduls(X, Y)),$
$sum(T2, C),$
$C \ \#{=}{<}\ 100.$

% Hier werden alle Prüfungsversuche gesucht; um anschließend zu prüfen, obverbrauchte CP nicht mehr als 100beträgt

.

%%
%% 4.3.1

$finde\_die\_Modul\_nr(pruefung\_vertiefungmodul\_nr : M, Z)$:-

$$M = Z.$$

% Mit dieser Klausel wird der Variable Z Modulnummer M zugeordnet.

%%
%% 4.3.2

$finde\_CP\_des\_Moduls(X, Y)$:-
$alle\_Vertiefungsmodule(W),$
$member(v\_modul\{modul\_nr : X, cp : Y\}, W).$

%%
%% 4.4

$modulabschlusspruefung\_bestanden\_Vertiefung(M, Z)$:-

$pruefungsversuche\_VM(W),$
$member(pruefung\_vertiefung\{modul\_nr : M,$
$pruefungszeit : Z, note : X\}, W),$
$not(X = 5.0),$
$not(X = nb).$

% Hier wird geprüft, ob Student zum Zeitpunkt Z das Modul M mit Erfolgabgeschlossen hat.

%%
%% 4.5

$zulassungsvoraussetzung\_Vertiefung(Voraussetzung, Zeit)$:-

$(Voraussetzung = 1- > voraussetzung\_HW\_PRG\_MOD\_DS(Zeit)$
$;$
$true),$
$(Voraussetzung = 2- > voraussetzung\_MOD\_DS(Zeit)$
$;$
$true),$
$(Voraussetzung = 3- > voraussetzung\_M1\_M2(Zeit)$
$; true),$
$(Voraussetzung = 4- > voraussetzung\_PRG\_HW(Zeit)$
$; true),$
$(Voraussetzung = 5- > voraussetzung\_PRG\_pr(Zeit)$
$; true),$
$(Voraussetzung = 6- > voraussetzung\_PRG(Zeit)$
$; true),$
$(Voraussetzung = 7- > voraussetzung\_MOD\_DS\_GL(Zeit)$
$; true).$

% Hier wird je nach Voraussetzung Klauseln aufgerufen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 4.5.1

$voraussetzung\_HW\_PRG\_MOD\_DS(Zeit)$:-

$pruefungsversuche\_BM(W),$
$member(pruefung\_basis\{modul\_nr : b\_HW,$
$pruefungszeit : Zeitpunkt, note : X\}, W),$
$not(X = 5.0),$
$Zeitpunkt\# =< Zeit, !.$

% Hier wird geprüft, ob das Modul b_HW abgeschlossen ist.

$voraussetzung\_HW\_PRG\_MOD\_DS(Zeit)$:-

$pruefungsversuche\_BM(W),$
$member(pruefung\_basis\{modul\_nr : b\_PRG,$
$pruefungszeit : Zeitpunkt, note : X\}, W),$
$not(X = 5.0),$
$Zeitpunkt \# =< Zeit, !.$

% Hier wird geprüft, ob das Modul b_PRG abgeschlossen ist.

$voraussetzung\_HW\_PRG\_MOD\_DS(Zeit)$:-

$pruefungsversuche\_BM(W),$
$member(pruefung\_basis\{modul\_nr : b\_MOD,$
$pruefungszeit : Zeitpunkt1, note : X1\}, W),$
$member(pruefung\_basis\{modul\_nr : b\_DS, pruefungszeit :$
$Zeitpunkt2, note : X2\}, W),$
$not(X1 = 5.0),$
$not(X2 = 5.0),$
$Zeitpunkt1$ #=< $Zeit,$
$Zeitpunkt2$ #=< $Zeit.$

% Hier wird geprüft, ob beide Module (b_MOD und b_DS)bereits abgeschlossen sind.

%%
%% 4.5.2

$voraussetzung\_MOD\_DS(Zeit)$:-

$pruefungsversuche\_BM(W),$
$member(pruefung\_basis\{modul\_nr : b\_MOD,$
$pruefungszeit : Zeitpunkt1,$
$note : X1\}, W),$
$member(pruefung\_basis\{modul\_nr : b\_DS,$
$pruefungszeit : Zeitpunkt2, note : X2\}, W),$
$not(X1 = 5.0),$
$not(X2 = 5.0),$
$Zeitpunkt1$ #=< $Zeit,$
$Zeitpunkt2$ #=< $Zeit.$

% Hier wird geprüft, ob beide Module (b_MOD und b_DS) bereits abgeschlossensind.

%%
%% 4.5.3

$voraussetzung\_M1\_M2(Zeit)$:-

$voraussetzung\_HW\_PRG\_MOD\_DS(Zeit),$
$pruefungsversuche\_BM(W),$
$member(pruefung\_basis\{modul\_nr : b\_M1,$
$pruefungszeit : Zeitpunkt1, note : X1\}, W),$
$member(pruefung\_basis\{modul\_nr : b\_M2,$
$pruefungszeit : Zeitpunkt2, note : X2\}, W),$

$not(X1 = 5.0)$,
$not(X2 = 5.0)$,
$Zeitpunkt1 \ \# =< Zeit$,
$Zeitpunkt2 \ \# =< Zeit$.

% Hier wird geprüft, ob beide Module (b_M1 und b_M2) bereits abgeschlossensind sowie die oben definierte Bedingung ($voraussetzung\_HW\_PRG\_MOD\_DS$)erfüllt ist.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 4.5.4

$voraussetzung\_PRG\_HW(Zeit)$:-

$pruefungsversuche\_BM(W)$,
$member(pruefung\_basis\{modul\_nr : b\_PRG, pruefungszeit : Zeitpunkt1,$
$note : X1\}, W)$,
$member(pruefung\_basis\{modul\_nr : b\_HW, pruefungszeit : Zeitpunkt2,$
$note : X2\}, W)$,
$not(X1 = 5.0)$,
$not(X2 = 5.0)$,
$Zeitpunkt1 \ \# =< Zeit$,
$Zeitpunkt2 \ \# =< Zeit$.

% Hier wird geprüft, ob beide Module (b_PRG und b_HW) bereits abgeschlossen sind.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 4.5.5

$voraussetzung\_PRG\_pr(Zeit)$:-

$voraussetzung\_HW\_PRG\_MOD\_DS(Zeit)$,
$pruefungsversuche\_BM(W)$,
$member(pruefung\_basis\{modul\_nr : b\_PRG\_pr,$
$pruefungszeit : Zeitpunkt,$
$note : X\}, W)$,
$not(X = 5.0)$,
$Zeitpunkt \ \# =< Zeit$.

% Hier wird geprüft, ob das Modul b_PRG_pr abgeschlossen ist.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 4.5.6

$voraussetzung\_PRG(Zeit)$:-

$pruefungsversuche\_BM(W),$
$member(pruefung\_basis\{modul\_nr : b\_PRG,$
$pruefungszeit : Zeitpunkt, note : X\}, W),$
$not(X = 5.0),$
$Zeitpunkt \#=< Zeit.$

% Hier wird geprüft, ob das Modul b_PRG abgeschlossen ist.

%%
%% 4.5.7

$voraussetzung\_MOD\_DS\_GL(Zeit)$:-

$pruefungsversuche\_BM(W),$
$member(pruefung\_basis\{modul\_nr : b\_MOD, pruefungszeit : Zeitpunkt1,$
$note : X1\}, W),$
$member(pruefung\_basis\{modul\_nr : b\_DS, pruefungszeit : Zeitpunkt2,$
$note : X2\}, W),$
$not(X1 = 5.0),$
$not(X2 = 5.0),$
$Zeitpunkt1 \#=< Zeit,$
$Zeitpunkt2 \#=< Zeit, !.$

% Hier wird geprüft, ob beide Module (b_MOD und b_DS) bereits abgeschlossen sind.

$voraussetzung\_MOD\_DS\_GL(Zeit)$:-

$pruefungsversuche\_BM(W),$
$member(pruefung\_basis\{modul\_nr : b\_GL, pruefungszeit : Zeitpunkt,$
$note : X\}, W),$
$not(X = 5.0),$
$Zeitpunkt \#=< Zeit.$

% Hier wird geprüft, ob das Modul b_GL abgeschlossen ist.

%%
%% Prädikat 5.

$das\_Gebiet\_Vertiefungsmodul\_abgesch(Module, CP\_Summe, Gebiete)$:-

$suche\_alle\_bestandene\_Module\_Vertiefung(Module),$

% Hier werden alle erfolgreiche Prüfungsversuche rausgefiltert.

$pruefe\_ob\_alle\_Vertiefungsmodule\_regelgemaess\_abgesch(Module)$,

% Hier wird geprüft, ob der Student die Module regelgemäß abgeschlossen hat.

$die\_CP\_Summe\_mindestens\_40(Module, CP\_Summe)$,

% Hier wird geprüft, ob Student mindestens 40 CP erzielt hat.

% Hier wird geprüft, ob Student ein Seminar und Praktikum
erfolgreichabgeschlossen hat.

$drei\_disjunkte\_Gebiete(S1, S2, S3)$,
$Gebiete = [S1, S2, S3]$.

/* Im Folgenden werden Klauseln definiert, die im Prädikat 5. benutzt wurden.*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % 5.1

$suche\_alle\_bestandene\_Module\_Vertiefung(List)$:-

$pruefungsversuche\_VM(W)$,
$(foreach(X1, W)$,
$fromto(L, Out, In, [])$ *do*
$X1 = pruefung\_vertiefungnote : 5.0 -> Out = In$
;
$Out = [X1|In])$,

% Hier werden alle erfolgreiche Prüfungsversuche rausgefiltert und in die Liste L geschrieben.

$(foreach(X2, L)$,
$foreach(Y2, List)$
$finde\_die\_Modul\_nr(X2, Y2))$.

% In der Liste List werden die Modulnummeraller Prüfungsversuche gesammelt.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5.2

$pruefe\_ob\_alle\_Vertiefungsmodule\_regelgemaess\_abgesch(Module)$:-

$(foreach(X, Module)$ *do*
$ein\_Wahlpflichtmodul\_abgesch\_Vertiefung(X))$.

%Hier wird geprüft, ob der Student die Module regelgemäß abgeschlossen hat.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5.3

$die\_CP\_Summe\_mindestens\_40(Liste, C)$:-

$(foreach(X, Liste),$
$foreach(Y, T)\ do$
$finde\_CP\_des\_Moduls(X, Y)),$

% In der Liste T werden die erzielten CP gesamelt.

$sum(T, C),$
$C\ \#>= 40.$

% Hier wird geprüft, ob die Summe der erzielten CP mindestens 40 beträgt.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5.4

$seminar\_und\_Praktikum\_enthalten(A, B)$:-

$pruefungsversuche\_VM(C),$
$member(pruefung\_vertiefung\{modul\_nr : A, note : T1\}, C),$
$T1\backslash = 5.0,$
$member(pruefung\_vertiefung\{modul\_nr : B, note : T2\}, C),$
$T2\backslash = 5.0,$
$alle_Vertiefungsmodule(G),$
$member(v\_modul\{modul\_nr : A, lehrform : prkt\}, G),$
$member(v\_modul\{modul\_nr : B, lehrform : smn\}, G).$

% Hier wird geprüft, ob der Student ein Seminar und ein Praktikum erfolgreichabgeschlossen hat.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5.5

$drei\_disjunkte\_Gebiete(D1, D2, D3)$:-

$suche\_alle\_bestandene\_Module\_Vertiefung(List),$

% Siehe oben.

$drei\_Wahlgebiete\_Vertiefung(V),$
$member(wahlgebiet\{gebiet\_1 : D1, gebiet\_2 : D2, gebiet\_3 : D3\}, V),$

% Hier werden die drei Wunschgebiete des Studenten gesucht.

$berechne\_CP\_Summe(List, D1, Summe\_1)$,

% Die Summe der im Gebiet $D1$ erzieltenCP werden in die Liste $Summe\_1$ geschrieben.

$berechne\_CP\_Summe(List, D2, Summe\_2)$,

% Die Summe der im Gebiet $D2$ ezieltenerzieltenCP werden in die Liste $Summe\_2$ geschrieben.

$berechne\_CP\_Summe(List, D3, Summe\_3)$,

% Die Summe der im Gebiet $D3$ erzielten CP werden in die Liste $Summe\_3$ geschrieben.

$Summen = [Summe\_1, Summe\_2, Summe\_3]$,

% Hier werden die Summen der drei Gebiete in die Liste *Summen* geschrieben.

$print(Summen), nl$,
$filter(Summen, 16, Rest1)$,
$filter(Rest1, 8, Rest2)$,
$filter(Rest2, 8, \_Rest3)$.

% Hier wird die Klausel $filter/3$ für die drei disjunkte Gebiete dreimalaufgerufen. Dabei sollen in den Gebieten jeweils 16, 8 und 8 CP erzielt werden.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5.5.1

$berechne\_CP\_Summe(L1, U, D)$:-

$(foreach(X, L1)$,
$fromto(L2, Out, In, [])$,
$param(U)$ do
$alle\_Vertiefungsmodule(V)$,
$(member(v\_modul\{modul\_nr : X, gebiet : U\}, V) - > Out = [X|In]$
;
$Out = In))$,

% Prüfungsleistungen im Gebiet U werden in die Liste $L2$ geschrieben.

$(foreach(A, L2)$,
$foreach(B, L3)$ do
$finde\_CP\_des\_Moduls(A, B))$,
$sum(L3, D)$.

% Die im Gebiet U erzielten CP werden addiert und die Summe wird in die Liste D geschrieben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5.5.2

```
filter([], _, _) :- false.
```

% Die leere Liste enthält kein Modul.

```
filter([X|T], Z, R) :-

    (X# >= Z,
    R = T)
    ;
    (X #< Z,
    append([X], W1, R),
    filter(T, Z, W1)).
```

% Hier wird rekursiv definiert, wie aus einer Liste von Zahlen die gesuchte Zahl rausgefiltert und wie der Rest für den nächsten Aufruf vorbereitet wird.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prädikat 6.

```
finde_best_moegliche_durchschnittsnote_Vertiefung(Durchschnittsnote,
                    Variationen) :-


findall(Durchschnittsnote, berechne_Durchschnittsnote_Vertiefung(_, _, _, _,
    Durchschnittsnote), N),
```

% Mit diesem Prädikat werden alle mögliche Durchschnittsnoten berechnet und in die Liste N geschrieben.

```
    X is min(N),
```

% Der Variable X wird die kleinste Zahl zugeordnet.

```
    breal_min(X, Durchschnittsnote),
```

% Hier wird aus dem Interval X die Untere Grenze genommen und der Variable Durchschnitt zugeordnet.

```
    length(N, Variationen),
    print(Variationen), nl.
```

% Hier wird zur Kontrolle die Zahl der möglichen Durchschnittsnoten berechnet undausgegeben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prädikat 7.

$berechne\_Durchschnittsnote\_Vertiefung(CP\_Gesamt, List\_CP, CP\_Vertiefung,$
$CP\_Anteile, Durchschnittsnote)$:-

$suche\_alle\_bestandene\_Module\_Vertiefung(List),$

% Hier werden alle erfolgreich abgeschlossene Module gesucht.

$suche\_alle\_erzielte\_CP(List, List\_CP),$
$sum(List\_CP, CP\_Vertiefung),$

% Hier werden die CP der Module gesucht.

$suche\_alle\_erzielte\_Noten(List, List\_Noten),$

% Hier werden die zugehörigen Noten geschrieben.

$ordne\_CP\_Anteile\_zu(List\_CP, CP\_Anteile),$

% Hier wird jeder $CP - Zahl$ in der Liste $List\_CP$ ein CP-Anteil zugeordnet.

$sum(CP\_Anteile, CP\_Gesamt), \qquad CP\_Gesamt = 40,$

% Hier wird vorausgesetzt, dass die Summe der CP_Anteile genau 40 entspricht.

$gewichtung\_der\_Noten\_durch\_CP(List\_Noten,$
$CP\_Anteile, Gewichtung),$

% Alle nummerische Noten werden mit CP gewichtet.

$sum(Gewichtung, Total),$

% Bildung der Summe aller Produkte($Total$).

$bilde\_CP\_Summe\_der\_benoteten\_Module(List\_Noten, CP\_Anteile,$
$CP\_Summe),$

% Bildung der $CP - Summe$ ohne die CP der unbenoteten Module.

$/(Total, CP\_Summe, Durchschnittsnote),$

% Bildung von Durchschnittsnote für Basismodule

$die\_CP\_der\_Wunschgebiete\_erfuellt(Summe1, Summe2, Summe3, CP\_Anteile)$,

% Hier wird geprüft, ob in den Wunschgebieten 16, 8 bzw 8 CP erreicht wurde.

$zwischenergebnisse(List, List\_Noten, List\_CP, CP\_Anteile,$
$Summe1, Summe2, Summe3, Durchschnittsnote)$.

% Hier werden Zwischenergebnisse zur Kontrolle ausgegeben.

/* Imfolgenden werden Klauseln definiert, die im Prädikat 7. benutzt wurden.*/

%%
%% 7.1

$suche\_alle\_erzielte\_CP(List, List\_CP)$:-

$(foreach(X, List),$
$foreach(Y, List\_CP)\ do$
$finde\_CP\_des\_Moduls(X, Y))$.

% Hier werden die CP der Module gesucht.

%%
%% 7.2

$suche\_alle\_erzielte\_Noten(List, List\_Noten)$:-

$(foreach(X, List),$
$foreach(Y, List\_Noten)\ do$
$suche\_Pruefungsnote(X, Y))$.

% Hier werden alle Prüfungsnoten gesucht.

%%%% 7.2.1

$suche\_Pruefungsnote(X, Y)$:-

$pruefungsversuche\_VM(W)$,
$member(pruefung\_vertiefung\{modul\_nr : X, note : Y\}, W)$,
$not(Y = 5.0)$.

% Hier wird für ein Modul die zugehörige Note gesucht.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 7.3

ordne_CP_Anteile_zu(List_CP,CP_Anteile):-

    (foreach(X1,List_CP),
    foreach(Y1,CP_Anteile) do
    Y1 #:: [0..X1],
    indomain(Y1)).

% Das Prädikat indomain(Y1) ordnet Y1 einen Wert aus dem Wertebereich
[0..X1] zu.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 7.4

gewichtung_der_Noten_durch_CP(List_Noten,
                    CP_Anteile,Gewichtung) :-

    (foreach(X2,List_Noten),
    foreach(Y2,CP_Anteile),
    foreach(Z,Gewichtung) do
    real(X2)-> *(X2,Y2,Z)
    ;
    Z = 0).

% Alle numerischen Noten werden mit CP gewichtet.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 7.5

bilde_CP_Summe_der_benoteten_Module(List_Noten,
                    CP_Anteile,Summe) :-

    (foreach(X,List_Noten),
    foreach(Y,CP_Anteile),
    foreach(Z,T4)do
    real(X)-> Z = Y
    ;
    Z = 0),
    sum(T4,Summe).

% Die CP der nicht numerischen und für den Durchschnitt uninteressanten
Notenbekommen den Wert 0. Die CP der restlichen wird addiert.
```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 7.6

$die\_CP\_der\_Wunschgebiete\_erfuellt(Summe1, Summe2, Summe3,$
$CP\_Anteile)$:-

$suche\_die\_drei\_Wunschgebiete(D1, D2, D3),$

% Hier werden die Drei Wuschgebiete des Studenten den Variablen $D1, D2$und $D3$zugeordnet.

$suche\_alle\_bestandene\_Module\_Vertiefung(List),$

% Hier werden alle erfolgreich abgeschlossene Module gesucht.

$(foreach(X, List),$
$foreach(Y, Gebiete)$ do
$finde\_Gebiet\_des\_Moduls(X, Y)),$

% Hier wird das Gebiet eines jeden bestandenen Modul gesucht und in die Liste Gebiete geschrieben.

$summiere\_CP\_im\_Gebiet(Gebiete, CP\_Anteile, D1, T1),$
$summiere\_CP\_im\_Gebiet(Gebiete, CP\_Anteile, D2, T2),$
$summiere\_CP\_im\_Gebiet(Gebiete, CP\_Anteile, D3, T3),$

% Hier wird aus der Liste Gebiete alle CP-Anteile den Gebieten $D1$, $D2$ und $D3$in die Liste CP_in_D geschrieben.

$sum(T1, Summe1),$
$sum(T2, Summe2),$
$sum(T3, Summe3),$

% Hier werden CP_Summe der einzelnen Gebiete berechnet.

$Summen = [Summe1, Summe2, Summe3],$
$filter(Summen, 16, Rest1),$
$filter(Rest1, 8, Rest2),$
$filter(Rest2, 8, \_Rest3).$

% Hier wird die Klausel $filter/3$ für die drei disjunkte Gebiete dreimal aufgerufen. Dabei sollen in den Gebieten jeweils 16, 8 und 8 CP erzielt werden.

%%%% 7.6.1

$suche\_die\_drei\_Wunschgebiete(D1, D2, D3)$:-

$drei\_Wahlgebiete\_Vertiefung(V),$
$member(wahlgebiet\{gebiet\_1 : D1, gebiet\_2 : D2, gebiet\_3 : D3\}, V).$

% Hier werden die drei Wunschgebiete des Studenten gesucht der Variablen $D1, D2$ und $D3$ zugeordnet.

%%%%7.6.2

$finde\_Gebiet\_des\_Moduls(X, Y)$:-

$alle\_Vertiefungsmodule(W),$
$member(v\_modul\{modul\_nr : X, gebiet : Y\}, W).$

% Hier wird der Variabe Y das Gebiet des bestandenen Modul X zugeordnet.

%%%% 7.6.3

$summiere\_CP\_im\_Gebiet(Gebiete, CP\_Anteile, D, CP\_in\_D)$:-

$(foreach(X, CP\_Anteile),$
$foreach(Y, Gebiete),$
$foreach(Z, CP\_in\_D),$
$param(D)\ do$
$(Y = D- > Z = X$
$;$
$Z = 0)).$

% Hier werden alle CP-Anteile eines Gebiets D in die Liste CP_in_D geschrieben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 7.7

$zwischenergebnisse(List, List\_Noten, List\_CP, CP\_Anteile,$
$Summe1, Summe2, Summe3, Durchschnittsnote)$:-

$print('AlleModule :'),$
$print(List), nl,$
$print('AlleNoten :'),$
$print(List\_Noten), nl,$
$print('AlleCP :'),$
$print(List\_CP), nl,$
$print('AlleCP - Anteile :'),$
$print(CP\_Anteile), nl,$
$print(Summe1), nl,$

$print(Summe2), nl,$
$print(Summe3), nl,$
$print('Durchschnittsnote :'),$
$print(Durchschnittsnote), nl.$

% Hier werden Zwischenergebnisse zur Kontrolle ausgegeben.

A.6 Das Modul: abschlussmodul

:- *module*(*abschlussmodul*).

% Der Name des Modules wird definiert.

:- *export*(*bachelorarbeit_erfolgreich_abgesch*/2).
:- *use_module*(*student*).
:- *use_module*(*alle_Module*).
:- *use_module*(*basismodule*).
:- *use_module*(*vertiefungsmodule*).
:- *use_module*(*ergaenzungsmodule*).

% Hier werden Modul geladen, die in diesem Modul aufgerufen werden.

:- *lib*(*ic*). % Die ic-Bibliothek wird geladen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Prädikat 9.

das_Abschlussmodul_abgesch(*CP_bisher*, *Durchschnitt*, *CP_Gesamt*):-

*erzielte_CP_Basismodule_min_*80(*BM_Total*),
erzielte_CP_Vertiefung(*VM_Total*),
erzielte_CP_Ergaenzung(*EM_Total*),
*gesammt_CP_min_*100(*BM_Total*, *VM_Total*, *EM_Total*, *CP_bisher*),
bachelorarbeit_erfolgreich_abgesch(*Durchschnitt*, *CP_Gesamt*).

% Hier werden alle Voraussetzungen definiert, die für das Abschliessen des Abschlussmoduls notwendig sind.

%%% Ende des Prädikats

/* Im folgenden werden Klauseln definiert, die im Prädikat 9 benutzt wurden.*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 9.1

$erzielte\_CP\_Basismodule\_min\_80(BM\_Total)$:-

$schreibe\_alle\_erzielte\_CP\_basis(Liste),$
$sum(Liste, BM\_Total),$
$BM\_Total \#>= 80.$

% Hier wird geprüft, ob in Basismodulen mindestens 80 CP erreicht worden sind.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 9.2
$erzielte\_CP\_Vertiefung(VM\_Total)$:-

$suche\_alle\_bestandene\_Module\_Vertiefung(List),$
$pruefe\_ob\_alle\_Vertiefungsmodule\_regelgemaess\_abgesch(List),$

%Hier werden alle bestandene Module in Vertiefungsmodul gesucht.

$(foreach(X, List),$
$foreach(Y, T) do$
$finde\_CP\_des\_Moduls(X, Y)),$

% In der Liste T werden die erzielten CP gesamelt.

$sum(T, VM\_Total).$

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 9.4

$gesammt\_CP\_min\_100(A, B, C, D)$:-

$Alle\_Summen = [A, B, C],$
$(foreach(X, Alle\_Summen),$
$from\ to(Expr, S1, S2, 0)\ do$
$S1 = X + S2),$
$D \#= eval(Expr).$

% Hier werden die erzielten CP in den Gebieten Basis-,Vertiefung- und Ergänzungsgebietes (A, B, C) addiert und deren Summe der Variable D zugeordnet.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 9.5

$bachelorarbeit\_erfolgreich\_abgesch(Durchschnitt, CP\_Gesamt)$:-

$oberseminar\_abgeschlossen(CP\_os, Z0)$,

$pruefungsversuch\_BA(T)$,
$member(bachelorarbeit\{modul\_nr : M, noten : X,$
$pruefungszeit : Z1\}, T)$,

$das\_Abschlussmodul(G)$,
$member(a\_modul\{modul\_nr : M, cp : CP\_ba\}, G)$,
$berechne\_Durchschnitt\_AM(X, Z0, Z1, Durchschnitt)$,
$CP\_Gesamt\ is\ CP\_os + CP\_ba.$

%Hier wird geprüft, ob und mit welcher Note der Student die Abschlussarbeit bestanden hat

%%%%%%%%%%%%
%% 9.5.1

$oberseminar\_abgeschlossen(A, Z)$:-

$pruefungsversuch\_OS(W)$,
$member(oberseminar\{modul\_nr : B, note : N, pruefungszeit : Z\}, W)$,
N \$=< 4.0,
$das\_Abschlussmodul(Q)$,
$member(a\_modul\{modul\_nr : B, cp : A\}, Q)$.

%Hier wird geprueft, ob das Oberseminar bereits abgeschlossen ist.

%%%%%%%%%%%%
%% 9.5.2
$berechne\_Durchschnitt\_AM(X, Z0, Z1, Durchschnitt)$:-

$Z0\ \#=< Z1$,

%Hier wird geprüft, ob Oberseminar bereits abgeschlossen hat.

$length(X, K)$,
$1\#= (K\ \#= 2 or K\ \#= 3)$,

% Hier wird vorausgesetzt, dass es nur zwei oder drei Benotungen geben kann.

$(K = 2- >$
$(foreach(H, X)do$
$H\$=< 4.0)$
;
$true)$,

% Hier wird vorausgesetzt, dass nur dann zwei Benotungen ausreicehen, wenn keine der Beurteilungen eine 5.0 ist.

$(K = 3- >$
$X = [M1|Rest]$,
$Rest = [M2|\_]$,
$-(M1, M2, Diff)$,
$abs(Diff, Abb)$,
$1 \#= ((M1\ \$= 5.0$ and $M2\ \$=< 4.0)$ or $(M1\ \$=< 4.0$ and
$M2\$= 5.0)$ or $(Abb\$\ \$>= 2.0))$
;
$true)$,

% Hier wird vorausgesetzt, dass nur dann eine dritte Beurteilung notwendig ist, wenn nur eine der ersten Beurteilungen eine 5.0 ist oder wenn die Differenz dieser mindestens 2.0 beträgt.

$sum(X, L)$,
$/(L, K, Durchschnitt)$,
$Durchschnitt\ \$=< 4.0$.

% Hier wird die Durchschnittsnote berechnet.

A.7 Das Modul: student

:- $module(student)$.

% Der Name des Modules wird definiert.

:- $exportabsolvierte\_studienleistungen/1$.
:- $exportpruefungsversuche\_BM/1$.
:- $exportpruefungsversuche\_VM/1$.
:- $exportdrei\_Wahlgebiete\_Vertiefung/1$.
:- $exporterfolgreiche\_pruefungsversuche\_EM/1$.
:- $exportpruefungsversuch\_OS/1$.
:- $exportpruefungsversuch\_BA/1$.

% Hier werden alle Strukturen exportiert

:- $exportstruct(studienleistung(vtg, pruefungszeit, semester))$.
:- $exportstruct(pruefung\_basis(modul\_nr, pruefungszeit, note))$.
:- $exportstruct(pruefung\_vertiefung(modul\_nr, pruefungszeit, note))$.
:- $exportstruct(wahlgebiet(gebiet\_1, gebiet\_2, gebiet\_3))$.
:- $exportstruct(ergaenzungsmodul(modul\_nr, pruefungszeit))$.
:- $exportstruct(oberseminar(modul\_nr, note, pruefungszeit))$.
:- $exportstruct(bachelorarbeit(modul\_nr, noten, pruefungszeit))$.

% Hier werden alle Strukturdefinitionen exportiert.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

$absolvierte\_studienleistungen([$
$studienleistung\{vtg : prog1, pruefungszeit : 7, semester : ws\},$
$studienleistung\{vtg : prog2, pruefungszeit : 4, semester : ss\},$
$studienleistung\{vtg : edgi, pruefungszeit : 6, semester : ws\},$
$studienleistung\{vtg : hwr, pruefungszeit : 2, semester : ss\},$
$studienleistung\{vtg : gl2, pruefungszeit : 6, semester : ss\},$
$studienleistung\{vtg : gl1, pruefungszeit : 2, semester : ws\}$
$]).$

% Hier werden alle absolvierte Studienleistungen im Basismodul geschrieben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

$pruefungsversuche\_BM([$
$pruefung\_basis\{modul\_nr : b\_HW, pruefungszeit : 4, note : 1.0\},$
$pruefung\_basis\{modul\_nr : b\_PRG, pruefungszeit : 6, note : 2.0\},$
$pruefung\_basis\{modul\_nr : b\_MOD, pruefungszeit : 1, note : 3.0\},$
$pruefung\_basis\{modul\_nr : b\_DS, pruefungszeit : 1, note : 1.0\},$

$pruefung\_basis\{modul\_nr : b\_GL, pruefungszeit : 3, note : 1.0\},$
$pruefung\_basis\{modul\_nr : b\_PRG\_pr, pruefungszeit : 6, note : b\},$
$pruefung\_basis\{modul\_nr : b\_HWS\_pr, pruefungszeit : 4, note : b\},$
$pruefung\_basis\{modul\_nr : b\_M1, pruefungszeit : 2, note : 1.0\},$
$pruefung\_basis\{modul\_nr : b\_M2, pruefungszeit : 5, note : 1.0\}$
]).

% Hier werden alle Pruefungsversuche im Basismodul geschrieben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

$pruefungsversuche\_VM([$
$pruefung\_vertiefung\{modul\_nr : b\_BS, pruefungszeit : 5, note : 2.0\},$
$pruefung\_vertiefung\{modul\_nr : b\_VS\_PR, pruefungszeit : 7, note : b\},$
$pruefung\_vertiefung\{modul\_nr : b\_PR\_BS, pruefungszeit : 5, note : 1.3\},$
$pruefung\_vertiefung\{modul\_nr : b\_SIM\_PR, pruefungszeit : 5, note : b\},$
$pruefung\_vertiefung\{modul\_nr : b\_EAL\_BS, pruefungszeit : 1, note : 2.3\},$
$pruefung\_vertiefung\{modul\_nr : b\_BK1, pruefungszeit : 5, note : 2.7\}$
]).

% Hier werden alle Pruefungsversuche im Vertiefungsmodul geschrieben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

$drei\_Wahlgebiete\_Vertiefung([wahlgebiet\{gebiet\_1 : bkspp, gebiet\_2 : ani, gebiet\_3 : gdi\}]).$

% Hier Schreibt der Student seine drei Wahlgebiete, in der er verpflichtet ist 16,8, bzw. nochmal 8 CP zu erzielen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

$erfolgreiche\_pruefungsversuche\_EM([$
$ergaenzungsmodul\{modul\_nr : [b\_TL, b\_BK1], pruefungszeit : 5\}$
]).

% Hier werden alle erfolgreiche Prüfungsversuche im Ergänzungsmodul geschrieben.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

$pruefungsversuch\_OS([$
$oberseminar\{modul\_nr : b\_BA\_os, note : 2.3, pruefungszeit : 5\}$
]).

% Hier werden alle Versuche geschrieben, das Oberseminar im Abschlussmodul zumachen .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

$pruefungsversuch\_BA([$

$$bachelorarbeit\{modul\_nr : b\_BA\_ba, noten : [5.0, 2.0, 2.0], pruefungszeit : 5\}$$
$$]).$$

% Hier werden alle Versuche geschrieben, die Bachelorarbeit im Abschlussmodul zu machen.

A.8 Das Modul: simulation_Bachelor

:- $module(simulation\_Bachelor)$.

% Der Name des Modules wird definiert.

:- $use\_module(student)$.
:- $use\_module(alle\_Module)$.
:- $use\_module(basismodule)$.
:- $use\_module(vertiefungsmodule)$.
:- $use\_module(ergaenzungsmodule)$.
:- $use\_module(abschlussmodul)$.
:- $use\_module(anwendungsfachmodule)$.

% Hier werden alle Module geladen.

:- $lib(ic)$.

% Die ic-Bibliothek wird geladen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Klausel 10.

$bachelor\_abgeschlossen(Gesamt\_Durchschnitt, Gesamt\_CP)$:-

$berechne\_Durchschnittsnote\_Basismodule(Durchschnitt\_1, CP\_Basis, CP\_1)$,

% Hier wird die Durchschnittsnote und die Gesamt-CP Basismodule berechnet.

$finde\_best\_moegliche\_durchschnittsnote\_Vertiefung(Durchschnitt\_2, \_)$,
$berechne\_Durchschnittsnote\_Vertiefung(CP\_2, \_, CP\_Vertiefung, \_, \_)$,

% Hier wird die Durchschnittsnote und die Gesamt-CP für Vertiefungsmodule berechnet.

$anwendungsfachmodule\_abgeschlossen(Durchschnitt\_3, CP\_3)$,

% Hier wird die Durchschnittsnote und die Gesamt-CP für Abschlussmodul berechnet.

$bachelorarbeit\_erfolgreich\_abgesch(Durchschnitt\_4, CP\_4)$,

% Hier wird die Durchschnittsnote und die Gesamt-CP für Abschlussmodul berechnet.

$erzielte\_CP\_Ergaenzung(CP\_Ergaenzung),$
$erzielte\_CP\_insgesamt\_mindestens\_180([CP\_Basis, CP\_Vertiefung,$
$CP\_3, CP\_4, CP\_Ergaenzung], Gesamt\_CP),$

$L1 = [Durchschnitt\_1, Durchschnitt\_2, Durchschnitt\_3, Durchschnitt\_4],$
$L2 = [CP\_1, CP\_2, CP\_3, CP\_4],$
$(foreach(X, L1),$
$foreach(Y, L2),$
$foreach(Z, L3)do$
$*(X, Y, Z)$
$),$
$sum(L3, Gewichtung),$
$sum(L2, CP\_Gesamt),$
$/(Gewichtung, CP\_Gesamt, X),$
$abrunden(X, Gesamt\_Durchschnitt),$
$print(L3), nl,$
$print(L1), nl,$
$print(L2), nl.$

% Hier wird CP-gewichteter Gesamt-Durchschnitt für das Bachelor berechnet, in dem alle Durchschnittnoten im Basis-, Vertiefung-, Anwendungs- und Bachelorarbeitmodul betrachtet werden.

/* Im folgenden werden Prädikate definiert, die in Klausel 4. benutzt wurden.*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 10.1

$erzielte\_CP\_insgesamt\_mindestens\_180(Liste, Gesamt\_CP)$:-

$sum(Liste, Gesamt\_CP),$
$Gesamt\_CP \# >= 180.$

% Hier wird die gesamten erreichten CP in allen Modulen berechnet.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 10.2

$abrunden(X, Z)$:-

$*(X, 10, Y),$
$fix(Y, T),$
$/(T, 10, Z).$

% Mit Hilfe eingebauter Prädikate wird beim abrunden nur die erste Dezimalstelle nach dem Komma berücksichtigt

Anhang B

Prädikatenübersicht

B.1 Prädikate in: Basismodule

1. $ein\_Pflichtmodul\_basis\_abgesch(Modul, Note)$.
   1.1 $Das\_Modul\_ist\_ein\_Basismodul/2$,
   1.2 $zulassungsvoraussetzung\_erfuellt/1$,
      1.2.1 $einer\_der\_Studienleistungen\_vorher\_erbracht/3$,
      1.2.2 $studienleistung\_vorher\_erbracht\_HWS/1$,
      1.2.3 $studienleistung\_vorher\_erbracht\_M2/1$,
   1.3 $not\_endgueltig\_nb/1$,
   1.4 $modulabschlusspruefung\_bestanden\_basis/2$,

2. $das\_Gebiet\_Basismodul\_abgesch(Modul)$.
   2.1 $finde\_nr\_des\_Moduls\_BM/1$,
   2.2 $pruefe\_ob\_alle\_Basismodule\_abgesch/1$,

3. $berechne\_Durchschnittsnote\_Basismodule(Durchschnittsnote, CP\_Total, CP\_benot)$
   3.1 $schreibe\_alle\_erzielte\_Noten\_basis/1$,
   3.2 $schreibe\_alle\_erzielte\_CP\_basis/1$,
   3.3 $pruefe\_ob\_CP\_Summe\_min\_98/2$,
   3.4 $gewichte\_numerische\_Noten\_durch\_CP\_1/3$,
3.5 $bilde\_CP\_Summe\_der\_numerischen\_Noten/3$,

B.2 Prädikate in: Vertiefungsmodule

4. *ein_Wahlpflichtmodul_abgesch_Vertiefung*(*Modul*).
   4.1 *das_Modul_ist_ein_Vertiefungsmodul*/2,
   4.2 *nicht_bereits_2_mal_Pruefung_wiederholt*/1,
   4.3 *cp_Konto_nicht_ueberlastet*/1,
   4.4 *modulabschlusspruefung_bestanden_Vertiefung*/2,
   4.5 *zulassungsvoraussetzung_Vertiefung*/2,
      4.5.1 *voraussetzung_HW_PRG_MOD_DS*/1,
      4.5.2 *voraussetzung_MOD_DS*/1,
      4.5.3 *voraussetzung_M1_M2*/1,
      4.5.4 *voraussetzung_PRG_HW*/1,
      4.5.5 *voraussetzung_PRG_pr*/1,
      4.5.6 *voraussetzung_PRG*/1,
      4.5.7 *voraussetzung_MOD_DS_GL*/1,

5. *das_Gebiet_Vertiefungsmodul_abgesch*(*Module*, *CP_Summe*, *Gebiete*).
   5.1 *suche_alle_bestandene_Module_Vertiefung*/1,
   5.2 *pruefe_ob_alle_Vertiefungsmodule_regelgemaess_abgesch*/1,
   5.3 *die_CP_Summe_mindestens_40*/2,
   5.4 *seminar_und_Praktikum_enthalten*/2,
   5.5 *drei_disjunkte_Gebiete*/3,

6. *finde_best_moegliche_durchschnittsnote_Vertiefung*(*Durchschnitt*, *Variationen*).

7. *berechne_Durchschnittsnote_Vertiefung*(*CP_Gesamt*,-
List_CP, *CP_Vertiefung*, *CP_Anteile*, *Durchschnittsnote*)
   7.1 *suche_alle_erzielte_CP*/2,
   7.2 *suche_alle_erzielte_Noten*/2,
   7.3 *ordne_CP_Anteile_zu*/2,
   7.4 *gewichtung_der_Noten_durch_CP*/3,
   7.5 *bilde_CP_Summe_der_benoteten_Module*/3,
   7.6 *die_CP_der_Wunschgebiete_erfuellt*/4,
      7.6.1 *suche_die_drei_Wunschgebiete*/3,
      7.6.2 *finde_Gebiet_des_Moduls*/2,
      7.6.3 *summiere_CP_im_Gebiet*/4,
   7. 7 *zwischenergebnisse*/8,

B.3 Prädikate in: Ergänzungsmodul

8. *ergaenzungsmodul_abgeschlossen*(*Modul*, *CP_Total*).
   8.1 *voraussetzung_Ergaenzungsmodul*/1,
   8.2 *voraussetzung_wenn_X_tutorium*/2,
      8.2.1 *bereits_abgesch*/2,
   8.3 *erzielte_CP_Ergaenzung*/1,
      8.3.1 *berechne_CP_Total*/2,

B.4 Prädikate in: Abschlussmodul

9. *das_Abschlussmodul_abgesch*(*CP_bisher*, *Durchschnitt*, *CP_Gesamt*)
   9.1 *erreichte_CP_Basismodule_min_*80/1,
   9.2 *erreichte_CP_Vertiefung*/1,
   9.3 *erzielte_CP_Ergaenzung*/1, (Siehe EM 8.3)
   9.4 *gesammt_CP_min_*100/4,
   9.5 *bachelorarbeit_erfolgreich_abgesch*/2.
      9.5.1 *oberseminar_abgeschlossen*/2.
      9.5.2 *berechne_Durchschnitt_AM*/4.

B.5 Prädikate in: Bachelormodul

10. *bachelor_abgeschlossen*(*Gesamt_Durchschnitt*, *Gesamt_CP*).
   10.1 *abrunden*/2,
   10.2 *erzielte_CP_insgesamt_mindestens_*180/2.

Literaturverzeichnis

[Barták98] Roman Barták.
On-line Guide to Constraint Programming, Prague 1998, http://kti.mff.cuni.cz/∼bartak/constraints/, zuletzt besucht: 2006-02-24.

[Barták99] Barták, R.
Constraint Programming: In Pursuit of the Holy Grail in Proceedings of Week of Doctoral Students (WDS99)
Part IV, MatFyzPress, Prague, June 1999, S.555-564,
http://kti.ms.mff.cuni.cz/∼bartak/downloads/WDS99.pdf, zuletzt besucht: 2006-02-24.

[EiOh] Norbert Eisinger, Hans Jürgen Ohlbach.
Deduktionssysteme. Automatisierung des logischen Denkens,
http://www.dfki.de/%7Ehjb/Deduktionssysteme/II-Grundlagen-und-Beispiele.pdf, zuletzt besucht: 2006-02-24.

[hamb02] Prof. Dr. Manfred Kudlek.
Skript zur Vorlesung Formale Grundlagen der Informatik: Logik im Wintersemester 2002-2003
http://www.informatik.uni-hamburg.de/WSV/f1/2002/f1.html, Entscheidbarkeitsprobleme, zuletzt besucht: 2006-02-24.

[KI02] Prof. Klaus Indermark.
Skript zur Vorlesung Logikprogrammierung im Wintersemester 2001-2002,
http://www2.s-inf.de/Skripte/HS/Logik.2001-WS-Indermark.(NiWi).LP_WS_01-02.pdf, zuletzt besucht: 2006-02-24.

[KmPs98a] Kim Marriott and Peter J. Stuckey.
Programming with Constraints: an Introduction. MIT Press, 1998.

[KmPs98b] Kim Marriott and Peter J. Stuckey.
Programming with Constraints: an Introduction, Powerpoint Course material for use with the book. http://www.cs.mu.oz.au, 1998, zuletzt besucht: 2006-02-24.

[MSJ97] Mark Wallace, Stefano Novello, Joachim Schimpf.
ECL^{i}PS^e : A Platform for Constraint Logic Programming
http://eclipse.crosscoreop.com, August 1997, zuletzt besucht: 2006-02-24.

[MSS04] Prof. Dr. Manfred Schmidt-Schauß
Skript zur Vorlesung Deduktionssysteme: Grundlagen und Anwendungen im Sommersemester 2004 http://www.ki.informatik.uni-frankfurt.de, zuletzt besucht: 2006-02-24

[muen05] Alexandru Berlea.
Skript zur Vorlesung Programmiersprachen im Wintersemester 2005, 04.11.2005, http://www.seidl.informatik.tu-muenchen.de, zuletzt besucht: 2006-02-24.

[NTGJ95] Norbert E. Fuchs, Tobias Kuhn, Gérard Milmeister, Jody Weissmann.
Dokumente zur Vorlesung Formale Grundlagen der Informatik,
http://www.ifi.unizh.ch/req/courses/formale_grundlagen/fgi_ss05/dokumente.html, 06-Jul-2005 Institut für Informatik Universität Zürich, zuletzt besucht: 2006-02-24.

[PiP90] William F. Clocksin, C.S. Mellish.
Programmieren in Prolog. Springer-Verlag, 1990.

[PoE92] Ralf Cordes, Rudolf Kruse, Horst Langendörfer, Heinrich Rust.
Prolog. Eine methodische Einführung, 3., verbesserte Auflage. 1992

[Ram95] Ramin Yasdi.
Logik und Programmieren in Logik, Prentice Hall Verlag GmbH, 1995.

[suny05] C.R. Ramakrishnan.
Advanced Logic Programming Spring 2005, http://www.cs.sunysb.edu, zuletzt besucht: 2006-02-24.

[Sus05] Prof. Dr. Susanne Biundo-Stephan.
Skript zur Vorlesung: Einführung in die Künstliche Intelligenz. Constraint Satisfaction Suche im Wintersemester 2005/2006, Abteilung Künstliche Intelligenz, Universität Ulm, http://www.informatik.uni-ulm.de, zuletzt besucht: 2006-02-24.

[ThSl97] Thom Frühwirth und Slim Abdennadher.
Anwendungen Constraintbasierter Programmierung 1997,
http://citeseer.ist.psu.edu/169434.html, zuletzt besucht: 2006-02-24.

[ThSl99] Thom Frühwirth und Slim Abdennadher.
Folien Constraint-Logikprogrammierung, 1999,
http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/buch.html, zuletzt besucht: 2006-02-24.

[thom93] Thom Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, M. Wallace.
Constraint Logic Programming: An informal Introduction (1993),
http://citeseer.ist.psu.edu/30286.htm, zuletzt besucht: 2006-02-24

[Tsang02] Prof. Edward Tsang.
Constraint Satisfaction Tutorial, 18 December 2002,
http://cswww.essex.ac.uk/CSP/ConstraintSatisfactionTeaching/, zuletzt besucht: 2006-02-24.

[Uwe00] Uwe Schöning.
Logik für Informatiker, 5. Auflage Spektrum Akademischer Verlag, 2000.

[Van89] Pascal Van Hentenryck.
Constraint Satisfaction in Logic Programming, 1989, The MIT Press Cambridge, Massachussetts London, England