FACHHOCHSCHULE WEDEL
DIPLOMARBEIT

in der Fachrichtung
Wirtschaftsinformatik

Thema:

Erstellung eines automatischen Stundenplanprogramms

Eingereicht von:	Rebecca Stock Schulterblatt 15 20357 Hamburg Tel.: 040/432 919 82 E-Mail: `rebecca.stock@gmx.de`
Erarbeitet im:	Wintersemester 2001/2002
Fertiggestellt am:	25. Januar 2002
Betreuer:	Prof. Dr. Uwe Schmidt Fachhochschule Wedel Feldstraße 143 22880 Wedel Tel.: 04103/8048-45 E-Mail: `si@fh-wedel.de`

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Hintergrund der Arbeit

Die allgemeine Problematik bei der Aufstellung von Stundenplänen für Schulen aller Art und damit auch für Universitäten ist die Verteilung von Veranstaltungen auf Personen und die begrenzte Zahl von Räumen mit den entsprechenden Ressourcen und Zeiten.
Veranstaltungen mit sich überschneidenden Klassen, Kursen oder Hörergruppen dürfen nicht gleichzeitig stattfinden und Räume müssen für die zugeteilten Veranstaltungen groß genug sein.
Im Gegensatz zu allgemeinbildenden Schulen haben Universitäten ein besonderes Raumproblem, da sie sehr unterschiedlich große Hörergruppen unterbringen müssen.
An der Fachhochschule Wedel und der Physikalisch-Technischen-Lehranstalt (PTL) gelten die folgenden Rahmenbedingungen:

- **5 Vorlesungstage** und **7 Zeiteinheiten** je Tag
- **300 Vorlesungen**
- **50 Professoren** und **Assistenten**
- **1200 Studenten**
  - 6 Fachrichtungen in der Fachhochschule
  - 3 Fachrichtungen in der Berufschule
- **25 Räume**

Die Erstellung des Stundenplans für die Fachhochschule Wedel/PTL ist wegen der engen Verzahnung der Studiengänge in der Fachhochschule und der Berufsschule sehr arbeitsaufwendig.

Eine automatische Stundenplanerstellung würde den Arbeitsaufwand erheblich reduzieren. Diese Diplomarbeit soll dazu einen Beitrag leisten.
Zum Testen der Funktionsfähigkeit des Programms wird auf die in [dip99] zusammengetragenen Daten zurückgegriffen.

1.2 Problemstellung

Unter Einbeziehung der Vorarbeiten von André Rieck[1] und der Assistenz von Michael Nitz, Andreas Ehlers und Marko Lütgens[2] ist das Ziel der Diplomarbeit die Erstellung eines Programms zur Stundenplanerstellung, das flexibel, einfach zu bedienen und leicht erweiterbar ist.
Dabei gelten weiterhin folgende Einschränkungen:

- **Konfliktfreiheit**
  Ein Raum darf zu einer Zeiteinheit nur einmal belegt werden und ein Teilnehmer kann zu einer Zeiteinheit nur an einer Veranstaltung teilnehmen.
- **Erfüllung von Ressourcenanforderungen**
  Raumgröße und Ressourcen (z.B. Beamer, Projektoren) müssen den Anforderungen entsprechen.
- **Einhaltung von speziellen Nebenbedingungen**
  Die für Vorlesungen und Teilnehmer definierten zeitlichen Einschränkungen müssen eingehalten werden.
- **Vollständigkeit**
  Alle definierten Veranstaltungen müssen in dem zu erstellenden Plan platziert werden.

1.3 Komplexität des Problems

Das Stundenplanproblem gehört in die Problemklasse der `NP-vollständigen` Probleme. Das bedeutet, dass es einen nichtdeterministischen Algorithmus gibt, der das Problem in polynomineller Zeit löst.

[1]Siehe dazu [dip99].
[2]Siehe dazu [ass00].

Weitere Problemklassen sind `P`, was bedeutet, dass es einen deterministischen Algorithmus gibt, der das Problem in polynomineller Zeit löst, und `EXP`, was bedeutet, dass es einen Algorithmus gibt, der das Problem in exponentieller Zeit löst.

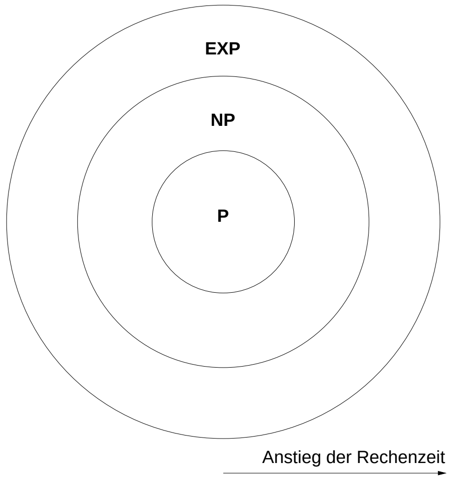

Abbildung 1.1: Abhängigkeiten zwischen `EXP`-, `NP`- und `P`-Problemen

Die Abbildung 1.1 zeigt, dass das `P`-Problem eine Teilmenge von `NP`-Problemen ist. `NP`-Probleme wiederum sind eine Teilmenge von `EXP`-Problemen.

Die Eigenschaft `vollständig` sagt aus, dass das Problem einen sehr hohen Schwierigkeitsgrad besitzt. Außerdem kann, wenn ein deterministischer Algorithmus für das Problem gefunden wird, damit nachgewiesen werden, dass die Problemklassen `P` und `NP` gleich sind.

Die `NP-Vollständigkeit` des Stundenplanproblems läßt sich beweisen, indem die einfachste Form des Problems auf das Graph-Coloring-Problem des Operations Research zurückgeführt wird. Das Graph-Coloring-Problem ist `NP-vollständig`[3].

Dies bedeutet, dass es nur einen nichtdeterministischen Algorithmus zur Lösung des Problems gibt. Somit muss die gesamte Menge aller möglichen Zuordnungen nach einer gültigen Lösung durchsucht werden.

[3]Ein Hinweis auf den vollständigen Beweis findet sich in [coo00], Seite 4.

Die Größe des Suchraums[4] wird durch

$$(Anzahl der Räume * Anzahl der Zeiten)^{Anzahl der Veranstaltungen} \tag{1.1}$$

definiert.
Bei den Daten der Fachhochschule Wedel/PTL (300 Veranstaltungen, 24 Räume und 35 Zeiten, in denen Veranstaltungen platziert werden können) ergibt sich eine Suchraumgröße von

$$(24 * 35)^{300}. \tag{1.2}$$

Wird eine Zeitspanne von `1 nS` für das Aufsuchen eines Punktes im Suchraum angenommen, dauert das Durchsuchen des gesamten Suchraums

$$(24 * 35)^{300} * 10^{-9} s = 2,22 * 10^{863} Tage. \tag{1.3}$$

Daraus folgt, dass eine ungerichtete Suche zeitlich nicht möglich ist.

1.4 Aufbau der Dokumentation

In Kapitel 2 werden die notwendigen Grundlagen erläutert. Dazu wird kurz auf `Constraint Programming` Grundlagen, logische Programmierung mittels `Prolog`, `Constraint Logic Programming` und die Expertensystementwicklungsumgebung `ECLiPsE` eingegangen.
Die allgemeinen Anforderungen an Stundenpläne und die speziellen Anforderungen an die Stundenplanerstellung an der Fachhochschule Wedel/PTL werden in Kapitel 3 beschrieben.
Der Entwurf eines Systems, das diese Anforderungen einbezieht, wird in Kapitel 4 dargestellt.
Die genaue Umsetzung wird in Kapitel 5 beschrieben. Dabei wird im Detail auf das entwickelte System eingegangen.
In Kapitel 6 wird auf die entwickelte konstruktive Suche eingegangen. Dabei wird die Vorgehensweise bei der Entwicklung in den Vordergrund gestellt.

[4] Formel entnommen aus [sem01].

Im letzten Kapitel werden die bei der Entwicklung des Stundenplanprogramms gewonnenen Erkenntnisse dargestellt. Außerdem werden Vorschläge zur Weiterentwicklung der Arbeit gegeben.

Im Anhang findet sich neben einer Installationsanleitung auch eine detaillierte Beschreibung der Wissensbasis, die dem Nutzer den Einstieg in das System erleichtern kann.

2 Eine Einführung in Constraint Logic Programming und das CLP-System `ECLiPSe`

2.1 Constraint Programming - Programmieren mit Einschränkungen

> „Constraint Programming is a generalization of logic programming, conversely logic programming is a special case of constraint programming[1].“

In letzter Zeit haben sich Constraints zu einem Forschungsbereich entwickelt, der Forscher aus vielen Gebieten zusammenbringt. Diese Gebiete sind unter anderem Künstliche Intelligenz, Programmiersprachen, Symbolic Computing und Computational Logic.

Constraint-Netzwerke und Constraint Satisfaction haben seit den 70er Jahren in der Künstlichen Intelligenz Eingang in die Forschung gefunden. Die systematische Nutzung von Einschränkungen bei der Programmierung hat in den 80er Jahren begonnen.

Die Programmierung eines Problems mit Einschränkungen beginnt mit der Definition der Anforderungen (Constraints) und dem Suchen möglicher Lösungswege. Zur Lösung können spezialisierte Constraint Solver eingesetzt werden.

Ein Constraint, eine Einschränkung, ist eine logische Relation zwischen diversen Unbekannten (oder Variablen), die einen Wert eines

[1]Fergus Henderson in `comp.lang.prolog` am 25. Oktober 2001.

gegebenen Wertebereiches (Domain) annehmen. Eine Einschränkung begrenzt die Werte, die eine Variable annehmen kann. Eine Einschränkung enthält somit Teilinformationen über die Variablen. Die Gesamtheit aller Einschränkungen beschreibt das Problem vollständig.
Einschränkungen haben folgende Eigenschaften[2]:

- Sie können partielle Informationen repräsentieren, d.h. dass mehrere Einschränkungen den Wert einer Variablen beschreiben können.
- Sie sind nicht gerichtet. Nimmt man zwei Variablen, `X` und `Y`, zwischen denen eine Einschränkungsbeziehung liegt, ist es für `Y` möglich `X` zu beeinflussen und umgekehrt.
- Sie sind additiv. Die Reihenfolge der Belegung von Variablen mit Einschränkungen hat keinen Einfluss auf die gefundene Lösung.
- Sie sind selten unabhängig. Typischerweise speichern Einschränkungen innerhalb der Einschränkungen gemeinsame Variablen.

Programmierung mit Einschränkungen bietet einen Kontext, um komplexe Suchprobleme zu lösen[3]. Diese Art der Programmierung kombiniert Methoden der Mathematik, der künstlichen Intelligenz und des Operations Research. Die Darstellung des Problems mit Hilfe der Einschränkungen führt zu kurzen, übersichtlichen und wartbaren Programmen, die sich leicht an wechselnde Anforderungen anpassen lassen. Wegen der kurzen Entwicklungszeiten und einfachen Erweiterbarkeit ist es möglich, verschiedene Lösungswege zu testen und den geeignetsten für das Problem auszuwählen. Des Weiteren können komplexere Probleme angegangen werden, ohne dass die Programmieraufgabe unhandlich wird.
Zur Zeit gibt es zwei unterschiedliche Ansätze zur Lösungsfindung eines Problems mit Einschränkungen.

2.1.1 Constraint Satisfaction

Constraint Satisfaction entstand aus der Forschung in Bereichen der Künstlichen Intelligenz und der Computergraphik. Beschrieben wird ein `Constraint Satisfaction Problem (CSP)` durch

[2] In Anlehnung an [bar98].
[3] Siehe dazu [wal95], Seite 1.

- eine endliche Menge von Variablen,

$$X = x_1..x_n \tag{2.1}$$

- eine Funktion, die jede Variable `x` mit einem endlichen Wertebereich `D`, der Domain, in Verbindung setzt und
- eine endliche Menge von Einschränkungen.

Jede Einschränkung grenzt die Kombination der Werte ein, die eine Menge von Variablen annehmen kann. Die Lösung eines `CSP`s ist die Zuweisung der Werte des Wertebereiches zu den Variablen. Die Aufgabe ist es, eine oder alle Lösungen zu finden.

2.1.2 Constraint Solving

Der Unterschied zwischen Constraint Solving und Constraint Satisfaction liegt in der Größe des Wertebereichs der Variablen. Statt der Nutzung eines endlichen Wertebereiches wird bei Constraint Solving hierbei ein unendlicher Wertebereich herangezogen. Auch ist es möglich, komplexere Constraints (nichtlineare Gleichheiten) zu hinterlegen. Somit nutzt das Constraint Solving mathematische und numerische Methoden, um Constraints zu lösen und greift nicht wie Constraint Satisfaction auf Kombination und Suche zurück.

2.2 Logische Programmierung mit PROLOG

> „A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning[4].“

In den 80er Jahren entwickelte sich ein neuer Zweig auf dem Gebiet der Programmierung, die logische Programmierung. Die bekannteste

[4]Entnommen aus [ste97], Seite 9.

Sprache dieses Zweiges ist PROLOG[5], entwickelt von Colmerauer in Marseille und Kowalski in Edinburgh.

Logische Programmierung unterscheidet sich von konventioneller Programmierung sowohl stilistisch als auch berechnungstechnisch, da bei logischer Programmierung das Problem deklarativ beschrieben und durch Schlussfolgerungen gelöst wird.

In der Literatur wird von der Trennung zwischen Kompetenz und Leistungsfähigkeit bzw. Logik und Kontrolle[6] gesprochen. So bieten logische Programmiersprachen die Möglichkeit, zuerst die Kompetenz, das Was, und erst danach, wenn dieses noch notwendig erscheint, die Leistungsfähigkeit, das Wie, zu betrachten.

PROLOG selbst ist eine prädikative Programmiersprache. In imperativen Programmiersprachen (z.B. PASCAL) wird ein gegebenes Problem dadurch gelöst, dass der Programmierer eine Folge von Anweisungen erstellt, durch deren Ausführung die Lösung gefunden wird. In PROLOG formuliert der Programmierer stattdessen sein eigenes Wissen über das Problem, und der Computer versucht mit Hilfe dieses Wissens, selbstständig eine Lösung des Problems zu finden.

In der Sprache der Logik besteht das Wissen aus einer Menge von wahren Aussagen und Regeln, die wahre Aussagen in wahre Aussagen überführen. Gibt der Benutzer eine Behauptung ein, so versucht das PROLOG-System, diese auf der Grundlage des Wissens zu beweisen.

Die Lösung einer gegebenen Anfrage findet das PROLOG-System durch das Backtracking-Verfahren, indem es alle Möglichkeiten solange ausprobiert, bis es eine Lösung findet oder bis es keine weiteren Möglichkeiten mehr gibt.

PROLOG wird wegen der Wissensverarbeitung vornehmlich in Bereichen der Künstlichen Intelligenz und der Expertensysteme und zur Entwicklung von Prototypen eingesetzt, jedoch ist PROLOG wegen seiner allgemeinen Konzepte nicht auf diese Bereiche beschränkt.

Backtracking

Backtracking ist die Bezeichnung für ein `Trial-and-Error-Verfahren`, das bei der Lösungssuche versucht, über Teillösungen eines Problems systematisch zu einer Gesamtlösung zu gelangen. Falls auf diesem Weg

[5] **PRO**gramming in **LOG**ic

[6] Siehe dazu [fru93], Seite 1.

eine Gesamtlösung nicht mehr möglich ist und das System in einer „Sackgasse“ endet, werden einer oder mehrere der letzten Schritte rückgängig gemacht. Von dieser Teillösung aus wird dann versucht, auf einem anderen Weg eine Gesamtlösung zu finden. Das Zurücknehmen von Schritten wird so lange wiederholt bis eine Lösung gefunden wird oder das Programm erkennt, dass das Problem nicht zu lösen ist. Die Möglichkeit, in Sackgassen zu laufen und aus ihnen wieder herauszufinden, zeichnet das Backtracking-Verfahren aus.

Viele Probleme lassen sich mit Hilfe des Backtracking-Verfahrens lösen, so das N-Dame-Problem, die Lösungssuche bei PROLOG und allgemein `NP-vollständige` Probleme. Da die Rechenzeit exponentiell mit der Suchtiefe anwächst, eignen sich Backtracking-Verfahren nur, wenn durch Zusatzbedingungen (z.B. Constraints) vorab möglichst viele Sackgassen ausgeschlossen werden.

Beispiel - Wie kommt die Maus zum Käse?[7]

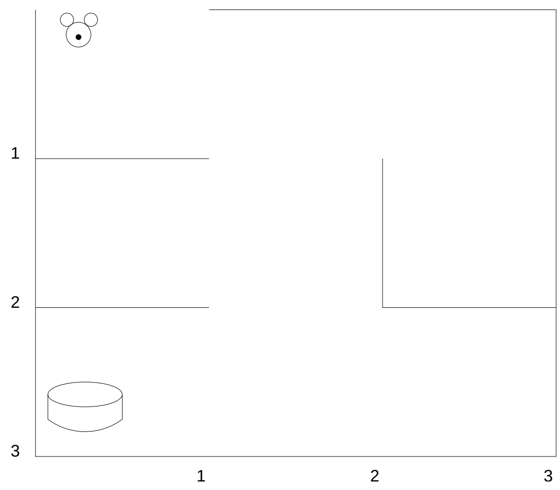

Abbildung 2.1: Labyrinth

[7]In Anlehnung an [dud93], Seite 51 f.

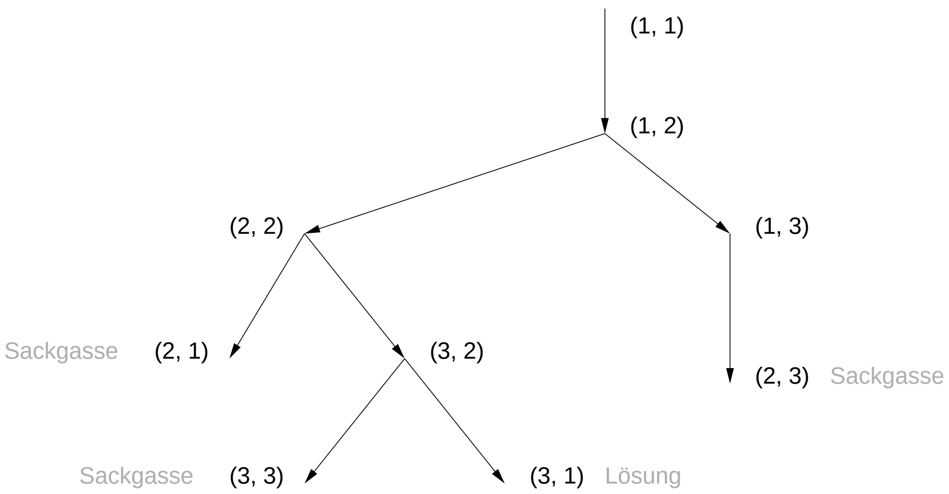

Abbildung 2.2: Mögliche Wege im Labyrinth

Abbildung 2.2 zeigt als Baum die möglichen Wege in dem in Abbildung 2.1 dargestellten Labyrinth. Die Knoten des Baumes entsprechen der Position der Maus im Labyrinth. In einer Sackgasse ermöglicht ein Backtracking-Verfahren, zum letzten korrekten Schritt zurückzukehren, um von dort nach einer Lösung zu suchen.

2.3 Constraint Logic Programming - CLP

> „Were you to ask me which programming paradigm is likely to gain most in commercial significance over the next 5 years I'd have to pick Constraint Logic Programming (CLP), even though it's perhaps currently one of the least known and understood.[8]"

Mit der Einbettung von Einschränkungen (constraints) in eine Logikprogrammiersprache ist es möglich, komplexe Probleme schnell und effizient zu lösen.

CLP-Sprachen vereinigen die Vorteile der Logikprogrammiersprachen mit denen von Constraintlösern. Die Kombination von lokaler Suche mit Constraintbehandlung ist besonders bei der Lösung kombinatorischer Probleme mit exponentieller Komplexität sehr nützlich.

[8]Entnommen aus [pou95], Seite 1.

Die erhöhte Deklarativität verringert den Programmieraufwand, ohne die Effizienz zu beeinträchtigen.

Das Lösen von Constraints soll das `generate-and-test`-Paradigma durch die Verkleinerung des Suchraumes vermeiden. Ziel ist es nicht, in einem vollständigen Suchbaum durch Tiefensuche eine gültige Lösung zu finden. Vielmehr geht es darum, den Suchbaum so zu gestalten, dass im günstigsten Fall nur gültige Lösungen erzeugt und damit die Einschränkungen gelöst werden. Die Realisierung dieser Methode wird als `constrain-and-generate`-Paradigma bezeichnet.

In dieser Diplomarbeit wird `Constraint Logic Programming` in Verbindung mit endlichen Wertebereichen (finite domains) verwendet. Ein endlicher Wertebereich ist eine endliche Menge von nicht-negativen Integerwerten.

Ein Problem mit endlichen Wertebereichen enthält eine endliche Menge `P` von bezeichnungsfreien Einschränkungen, so dass `P` mindestens eine Domaineinschränkung für jede der Variablen in `P` enthält. Die Lösung eines Problems mit endlichen Wertebereichen ist eine Variablenzuweisung, die jede der Einschränkungen erfüllt.

Probleme, die mittels `CLP(FD)` definiert werden, besitzen eine endliche Anzahl von Lösungen oder keine Lösung.

Diese Vorgehensweise lässt sich gut an den folgenden Beispielen zeigen.

2.4 Das N-Dame-Problem

Wie muss man N Damen auf einem Schachbrett der Maße `N * N` platzieren, damit sich keine zwei Damen gleichzeitig bedrohen.

Zuerst wird eine Lösung vorgestellt, die ohne Constraint-Techniken auskommt und auf das `generate-and-test`-Paradigma zur Lösung zurückgreift. Danach wird das Problem unter Einsatz von Einschränkungen und Suchraumverkleinerung gelöst.

2.4.1 Das N-Dame-Problem gelöst in PROLOG

```
damen(Anzahl, Spielfeld) :-
     % erstellen einer Liste mit N Werten
     bereich(1, Anzahl, Liste),
     % belegen aller Werte der Liste
     permutation(Liste, Spielfeld),
```

```
    % feststellen, ob die Belegung korrekt ist
    sicher(Spielfeld).

sicher([]).
sicher([Kopf|Rest]) :-
    sicher(Rest),
    not bedroht(Kopf, Rest).

bedroht(Element, Liste) :-
    bedroht(Element, 1, Liste).

bedroht(Element, Anzahl, [Kopf|_]) :-
    (
       Element is Kopf + Anzahl
      ;
       Element is Kopf - Anzahl
    ).
bedroht(Element, Anzahl, [_|Rest]) :-
    NeueAnzahl is Anzahl + 1,
    bedroht(Element, NeueAnzahl, Rest).

permutation([], []).
permutation([Kopf|Rest], Ergebnis) :-
    permutation(Rest, ZwErgebnis),
    findeEinElement(Kopf, ZwErgebnis, Ergebnis).

% findeEinElement sucht aus den noch vorhandenen Werten
% einen aus und fuegt diesen der Permutation zu
findeEinElement(Element, Liste, Ergebnis) :-
    waehle(Element, Ergebnis, Liste).

waehle(Element, [Element|Rest], Rest).
waehle(Element, [AnderesElement|Rest1],
      [AnderesElement|Rest2]) :-
    waehle(Element, Rest1, Rest2).

% bereich erzeugt eine Liste mit MaxAnzahl-Elementen,
% dabei wird 1 Element pro Zeile erstellt,
% diesem Wert wird im Verlauf des Programmes der
% Standpunkt der Dame zugewiesen
bereich(AktAnzahl, MaxAnzahl, [AktAnzahl|Rest]) :-
    AktAnzahl < MaxAnzahl,
    NeueAktAnzahl is AktAnzahl + 1,
    bereich(NeueAktAnzahl, MaxAnzahl, Rest).
bereich(MaxAnzahl, MaxAnzahl, [MaxAnzahl]).
```

Das Prädikat `bereich` erzeugt eine Liste der Zahlen zwischen 1 und `Anzahl`. Dann beginnt der `generate-and-test`-Zyklus. Das `permutation`-Prädikat erzeugt eine Permutation des `Spielfeld`es, wobei `sicher(Spielfeld)` überprüft, ob es sich um eine Lösung des Problems handelt. Dieses Prädikat ist wahr, wenn die gefundene Permutation eine korrekte Aufstellung der Damen ist. Da zwei Damen nicht in dieselbe Spalte oder Zeile gestellt werden, braucht das Prädikat nur zu prüfen, ob sich zwei Damen diagonal bedrohen. Eine Liste von Damen ist `sicher`, wenn die vom Rest der Liste repräsentierten Damen sicher sind und die vom Kopf der Liste repräsentierte Dame keine der anderen Damen bedroht. Die Definition `bedroht(Element, Liste)` verwendet eine Formulierung der Beziehungen von Diagonalen. Eine Dame ist auf derselben Diagonalen wie eine zweite Dame, die `Anzahl` Spalten entfernt ist, wenn die Zeilennummer dieser Dame `Anzahl` Einheiten größer oder `Anzahl` Einheiten kleiner ist als die Zeilennummer der ersten Dame. Dies wird durch die erste Regel `bedroht/3` ausgedrückt. `bedroht(Element, Liste)` bedeutet, dass eine Dame `Element` irgendeine Dame in `Liste` bedroht. Die Diagonalen werden iterativ geprüft bis das Ende des Brettes erreicht ist. Bedrohen sich die Damen nicht untereinander, ist eine Lösung gefunden und das Programm endet.

2.4.2 Das N-Dame-Problem gelöst mit Constraint-Techniken

Das Beispiel ist in Anlehnung an [bra01], Seite 344 entwickelt worden.

```
:- lib(fd).

damen(Anzahl, Spielfeld) :-
    % erstellen der benoetigten Variablen
    length(Spielfeld, Anzahl),
    % zuweisen des Wertebereiches
    Spielfeld :: 1..Anzahl,
    % belegen der Variablen mit Einschraenkungen
    (
       fromto(Spielfeld, [EineSpalte|Spalten], Spalten, [])
    do
       (
        foreach(NochEineSpalte, Spalten),
        count(Distanz, 1, _),
        param(EineSpalte)
```

```
        do
         sicher(EineSpalte, NochEineSpalte, Distanz)
        )
    ),
    % berechnen einer Loesung
    labeling(Spielfeld).

sicher(EineSpalte, NochEineSpalte, Distanz) :-
    EineSpalte ## NochEineSpalte,
    NochEineSpalte - EineSpalte ## Distanz,
    EineSpalte - NochEineSpalte ## Distanz.
```

Das Spielfeld wird in diesem Beispiel über das `length`-Prädikat erzeugt. Danach wird über `Spielfeld :: 1..N` jedem Element der Liste der Wertebereich zugewiesen. In der folgenden Schleife werden die Einschränkungen - zwei Damen dürfen sich nicht attackieren - über das Prädikat `sicher(EineSpalte, NochEineSpalte, Distanz)` den einzelnen Listenelementen zugewiesen. `sicher(EineSpalte, NochEineSpalte, Distanz)` verbietet, dass zwei Damen in einer Spalte bzw. auf einer Diagonalen stehen. Danach beginnt mit Hilfe des `labeling(Spielfeld)` der Lösungsfindungsprozess. `labeling/1` fügt die Standardsuchfunktion aus `ECLiPSe` ein.

Das Beispiel zeigt deutlich den Aufbau von Constraint-Programmen[9]:

```
start :-
    aufbauDerVariablenUndEinschraenkungen(Variablen),
    loesungsFindung(Variablen),
    ausgabeDerLoesung(Variablen).
```

Der `aufbauDerVariablenUndEinschraenkungen` sollte deterministisch sein. Alle nicht-deterministischen Teile gehören in die `loesungsFindung`. Die `ausgabeDerLoesung` sorgt für die lesbare Darstellung der Lösung.

2.5 Das CLP-System ECLiPSe

`ECLiPSe` (ECRC Common Logic Programming System) ist ein Prolog-System mit zusätzlichen Erweiterungen für die Constraint-Programmierung und einigen constraint-spezifischen Bibliotheken.

Eine dieser `ECLiPSe`-Bibliotheken ist die `fd`-Library. Sie fügt Constraints über endliche Wertebereiche ein. Diese Wertebereiche können

[9] Joachim Schimpf, IC-Parc, Imperial College in der `ECLiPSe-User`-Mailingliste

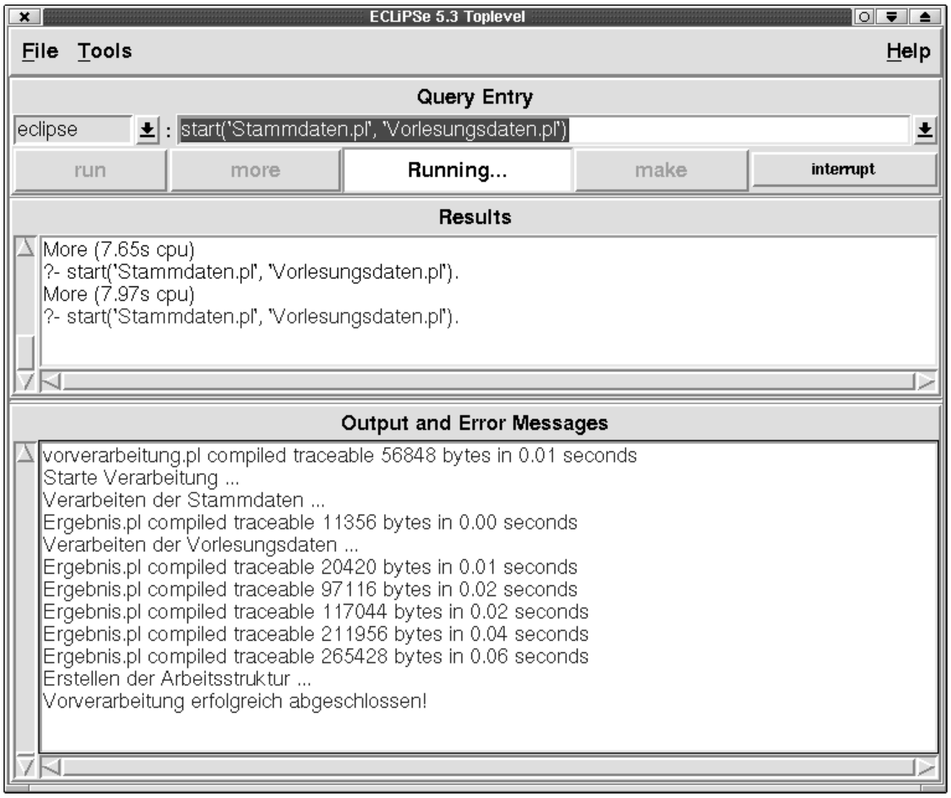

Abbildung 2.3: Die Expertensystementwicklungsumgebung `ECLiPSe`

z.B. Integer-Werte, Atome und Fliesskommazahlen sein. In der Bibliothek gibt es eine Reihe von low-level Prädikaten, die es dem Benutzer ermöglichen, Variablen und ihre Wertebereiche zu verarbeiten. Außerdem ist es möglich, eigene Constraint-Prädikate zu schreiben.

Eine andere `ECLiPSe`-Bibliothek ist die `repair`-Library. Sie unterstützt die Verwendung von Constraints in Verbindung mit Repair-Constraints. Programme, die mit Repair-Constraints arbeiten, lassen Inkonsistenzen innerhalb des Systems zu.

Eine Suche mit Hilfe der `repair`-Bibliothek kann mit dynamischen Problemen umgehen. Dynamisch heißt in diesem Fall, dass sich Anforderungen nach einer ersten Lösung ändern können. Diese Änderungen erfolgen aus Gründen, die nicht im System erfasst worden sind bzw. erfasst werden können. Der Nutzer kann nun neue Constraints hinzufügen, die die bisherige Lösung ausschließen. So kann eine neue

Lösung gefunden werden, die den Anforderungen besser gerecht wird.

In `ECLiPSe` wird dieses Konzept mit Hilfe von `tentative values`, der vorläufigen Wertebelegung, umgesetzt. Während des `labeling`-Prozesses werden den einzelnen Elementen vorläufige Werte zugewiesen, die bei einer Anforderungsänderung überprüft werden. Sollten die vorläufig zugewiesenen Werte die neu gestellten Anforderungen nicht mehr erfüllen, werden sie in das `conflict set` eingefügt. Diese Konfliktmenge verwaltet die Wertebelegungen, die an Konflikten beteiligt sind. Die Lösungsfindung ist erst beendet, wenn die Konfliktmenge keine Elemente mehr enthält.

Mit Hilfe der `repair`-Bibliothek lassen sich unter anderem lokale Suchverfahren wie `Great Deluge`, der `Sintflutalgorithmus`[10], einbauen.

Die lokale Suche mittels der `repair`-Bibliothek lässt sich sehr gut am Rucksackproblem, einem `NP-vollständigen` Problem, zeigen.

Gegeben sind ein Rucksack mit einer maximalen Tragfähigkeit `G` und `n` Gegenstände unterschiedlichen Gewichts und unterschiedlichen Wertes. Der Rucksack soll so bepackt werden, dass einerseits das Gesamtgewicht der eingepackten Gegenstände die Tragfähigkeit des Rucksacks nicht überschreitet und andererseits der Gesamtwert der Gegenstände möglichst groß ist. Die Lösung[11] verwendet den `Sintflutalgorithmus`, endliche Wertebereiche und die `repair`-Bibliothek.

```
:- lib(fd).
:- lib(repair).
```

[10] Dieser Algorithmus wurde 1990 veröffentlicht. Weitere Informationen finden sich in [due90]. Der Algorithmus rechnet relativ lange im Vergleich zu anderen Algorithmen seiner Klasse, ist aber leicht zu benutzen, da nur ein Parameter, der Regen, einstellbar ist.

```
Great Deluge Algorithmus zur Maximierung:

 waehle eine Anfangskonfiguration,
 waehle eine Regenmenge Regen > 0,
 waehle einen Wasserstand > 0,
 WIEDERHOLE:
     waehle eine neue Konfiguration,
     die eine kleine Aenderung der alten Konfiguration ist,
     berechne die Qualitaetsfunktion Q der neuen Konfiguration,
     WENN Q(neu) > Wasserstand
     DANN alte Konfiguration := neue Konfiguration,
     Wasserstand := Wasserstand + Regen,
 BIS lange keine Erniedrigung der Qualitaet,
ENDE
```

[11] In Anlehnung an [tut99], Seite 18 ff.

```
loeseRucksackProblem :-
     % fuer 100 Durchlaeufe
     between(1, 100, 1, Instanz),
     % lasse greatDeluge 1000-mal durchlaufen
     once greatDeluge(Instanz, 1.0, 1000, _).

greatDeluge(Id, Regen, MaxIter, Opt) :-
     initRucksackProblem(Id, Array, Profit, _, _),
     greatDeluge(Regen, MaxIter, Array, Profit, Opt).

greatDeluge(Regen, MaxIter, Array, Profit, Opt) :-
     % die einfachste Ausgangsloesung ist der
     % leere Sack
     generiereAusgangsloesung(Array),
     (
         % von 0 bis MaxIter
         fromto(0, I0, I1, MaxIter),
         % Anfangswasserstand ist 0
         fromto(0, WasserStand, NeuerWasserStand, _),
         % Anfangsoptimum ist 0,
         % das Optimum der Berechnung steht in Opt
         fromto(0, Opt2, Opt3, Opt),
         % in diesem Zusammenhang globale Variablen
         param(Array, Profit, Regen)
     do
         (
          tauscheZufaellig(Array),
          % repair-Bibliothek
          % auslesen des zugehoerigen Wertes
          Profit tent_get AktuellerProfit,
          AktuellerProfit > WasserStand,
          % repair-Bibliothek
          % auslesen der Variablen, die in einer
          % Konfliktmenge ueberwacht werden
          % -> steht fuer eine if-then-else-Anweisung
          conflict_constraints(cap,[])
         ->
          (
              AktuellerProfit > Opt2
          ->
              I1 = 0,
              % Fortschrittsausgabe
              write('Habe␣Loesung␣mit␣'),
              write(AktuellerProfit),
              writeln('␣gefunden.'),
              Opt3 = AktuellerProfit
```

```
          ;
              I1 is I0 + 1,
              Opt3 = Opt2
          )
         ;
          I1 is I0 + 1,
          Opt3 = Opt2
         ),
         NeuerWasserStand is WasserStand + Regen
     ).

initRucksackProblem(Id, Array, Profit, Profite, Gewicht) :-
     datenFuerRucksackProblem(Id, N, Profite, Gewicht, Fassungsvermoegen),
     length(Variablen, N),
     Variablen :: 0..1,
     % repair-Bibliothek
     % belge eine Variable als repair-Variable und
     % ueberwache sie auf ausloesende Konflikte
     Fassungsvermoegen #>= Gewicht * Variablen r_conflict cap,
     % repair-Bibliothek
     % festschreiben eines vorlaeufigen Wertes
     Profit tent_is Profite * Variablen,
     % berechnen des Profits je Gewichtseinheit
     (
         foreach(X, Variablen),
         foreach(P, Profite),
         foreach(W, Gewicht),
         foreach(ProfitProGewicht - X, PPGs)
     do
         ProfitProGewicht is P/W
     ),
     % erstellen eines sortierten Arrays, wo die
     % erfolgsversprechensden Element am Anfang stehen
     sort(1, >=, PPGs, SortiertePPGs),
     (
         foreach(_ - X, SortiertePPGs),
         foreach(X, SortierteVariablen)
     do
         true
     ),
     Array =.. [[]|SortierteVariablen].

tauscheZufaellig(Array) :-
     functor(Array, _, N),
     % suche einen zufaelligen Wert aus
     X is Array[random mod N + 1],
     % repair-Bibliothek
```

```
    % auslesen eines vorlaeufigen Wertes
    X tent_get Old,
    % war der Wert im Rucksack
    % wird er jetzt herausgenommen
    % und umgekehrt
    New is xor(Old, 1),
    % repair-Bibliothek
    % setzen eines vorlaeufigen Wertes
    X tent_set New.

generiereAusgangsloesung(Array) :-
    (
       foreacharg(X, Variablen)
    do
       % setze den Wert der Variablen auf
       % den kleinsten Wert der Domain
       % (nicht im Sack enthalten)
       mindomain(X, T),
       % repair-Bibliothek
       % setzen eines vorlaeufigen Wertes
       X tent_set T
    ).
```

Beispieldaten zur Ausführung des Programms finden sich im Anhang C.

ECLiPSe stellt noch weitere für das Problem relevante Bibliotheken zur Verfügung:

- ic
  Die ic-Bibliothek arbeitet ähnlich der fd-Bibliothek. In naher Zukunft soll die ic-Bibliothek alle CLP(X)-Bibliotheken ablösen.

- fd_global
  Sie vereint verschiedene globale Einschränkungen über Listen mit endlichen Wertebereichen.

- fd_search
  Sie enthält einfache Suchroutinen, die verschiedene Verfahren verdeutlichen.

- fd_sets
  Diese Bibliothek enthält Löser für Integer-Mengen.

- branch_and_bound
  Die branch_and_bound-Bibliothek enthält einfache Methoden für

die Umsetzung einer konstruktiven Suche. Ein Optimierungsproblem wird dabei nach gewissen Regeln in Teilprobleme (`branch`) zerlegt. Die einzelnen Teilprobleme bilden einen Baum. Jedem Knoten innerhalb des Baumes wird ein Wert zugeordnet, der eine Schranke (`bound`) für den Wert der Zielfunktion des Optimierungsproblems darstellt.

3 Anforderungsanalyse

Bei der Anforderungsanalyse geht es darum, die zu erfüllenden Eigenschaften des Systems zu definieren. Dazu muss das Problem mit allen seinen Anforderungen erfasst werden. Die Analyse dient dann als Basis für die Entwicklung des Stundenplanerstellungssystems.

3.1 Anforderungen allgemein

3.1.1 Veranstaltungen

Die Veranstaltungen während eines Semesters ergeben sich aus den Studien- und Lehrplänen. Die Veranstaltungen sind entweder durch den Studienplan zwingend vorgeschrieben oder freiwillig. Zwingend vorgeschriebene Veranstaltungen einer Hörergruppe dürfen nicht parallel liegen. Veranstaltungen, an denen die Teilnahme freiwillig ist, können, aber müssen nicht parallel geplant werden.

An der Fachhochschule Wedel/PTL gibt es vier unterschiedliche Veranstaltungstypen.

- **Vorlesungen**
  Vorlesungen sind Veranstaltungen, die von einem Dozenten gehalten werden und Mindestanforderungen an Räume und andere Ressourcen stellen.

- **Praktika und Übungen**
  Praktika werden von einem oder mehreren Assistenten betreut und können nur in speziell für diese Veranstaltungen eingerichteten Räumen stattfinden.

- **Seminare**
  Seminare werden von einem Dozenten betreut, aber von Seminarteilnehmern gehalten.

- **Termin nach Absprache**
  Ein Termin nach Absprache ist eine Veranstaltung, deren Termin erst zu Beginn des Semesters durch alle teilnehmenden Personen gemeinsam bestimmt wird. Eine Platzierung und somit eine Berücksichtigung zur Zeit der Planerstellung ist nicht notwendig.

Eine Veranstaltung wird durch ihren Namen, ihre Teilnehmer, einschließlich der Dozenten und Assistenten, die von ihr benötigten Ressourcen und die benötigten Zeiteinheiten beschrieben.
Es kommt vor, dass eine Veranstaltung abwechselnd in zwei Räumen stattfinden muss. Außerdem kann die Gesamtteilnehmeranzahl die Hörsaalgröße überschreiten. Solche Veranstaltungen müssen trotzdem im Plan platzierbar sein.

3.1.2 Hörergruppen

Dozenten und Assistenten werden über ihre Namenskürzel und Studenten über ihre Studienrichtung in Verbindung mit dem Fachsemester definiert.

Das Untergruppenkonzept

Es gibt Veranstaltungen, an denen nicht alle Teilnehmer einer Hörergruppe teilnehmen, Übungen und Wahlblöcke wie Dienstleistungsmanagement und Informatik für die Wirtschaftsinformatiker. Die Veranstaltungen eines Wahlblocks oder einer Übung dürfen parallel liegen. Eine Überschneidung mit anderen Veranstaltungen dieser Hörergruppe muss natürlich ausgeschlossen sein. Durch diese parallel planbaren Untergruppen kann das Gesamtvolumen des Stundenplanes für die einzelnen Teilnehmer reduziert werden.
Eine weitere Unterteilung (Untergruppen von Untergruppen) verbietet sich, da dann keine Feststellung über die parallele Planbarkeit getroffen werden kann.

3.1.3 Räume

Die Räume an der Fachhochschule Wedel/PTL lassen sich in

- Hörsäale,
- Rechenzentren,
- Labore und
- Seminarräume

unterteilen. Diese Einteilung ist für die Stundenplanerstellung aber nur von geringer Bedeutung. Die Entscheidung, ob eine Veranstaltung in einem bestimmten Raum stattfinden kann oder nicht, beruht vielmehr auf den Ressourcenanforderungen einer Veranstaltung. Ein Raum wird somit über seinen Namen, seine Größe und sein Ressourcenangebot beschrieben. Da sich Zahl, Art und Ausstattung der Räume ändern können, müssen Räume im System frei definierbar sein.

3.1.4 Tage und Zeiten

Vorlesungstage und Zeiten sind durch ihren Namen im System bekannt. Die Tage gleichen sich in ihrer Einteilung in Zeiteinheiten. Einzelne Vorlesungen können sich über mehrere Zeiteinheiten eines Tages, nicht aber über mehrere Tage erstrecken.

3.1.5 Harte Einschränkungen und Wünsche

3.1.5.1 Die harten Einschränkungen

Es gibt Einschränkungen, deren Nichteinhaltung zu einem ungültigen Stundenplan führen.

- Pflichtveranstaltungen, deren Teilnehmer sich überschneiden, dürfen nicht parallel geplant werden.
- Räume dürfen in einer Zeiteinheit nicht doppelt belegt werden.
- Eine Veranstaltung darf nur in einem Raum stattfinden, der den Ressourcen- und Größenanforderungen entspricht.

- Mehrfachstunden müssen zeitlich und räumlich zusammenhängen[1].

Weiterhin muss das System Einschränkungen der Verfügbarkeit der Teilnehmer berücksichtigen, sowohl hinsichtlich der Tage als auch der Zeiten.

Einschränkungen von Teilnehmern, Räumen und Veranstaltungen müssen, solange sie sich nicht gegenseitig ausschließen, frei miteinander kombinierbar sein.

3.1.5.2 Die Wünsche

Anders als harte Einschränkungen müssen Wünsche, sogenannte weiche Einschränkungen, in einem gültigen Plan nicht umgesetzt werden. Vielmehr wird durch die Berücksichtigung der Wünsche ein besserer[2] Plan erstellt.

Einige harte Einschränkungen sollten auch als Wünsche genutzt werden können.

Ein Teilnehmer könnte angeben, dass eine seiner Veranstaltungen lieber als Doppelstunde gehalten werden sollte. Als Wunsch definiert, ist die Umsetzung als Doppelstunde nicht zwingend notwendig, steigert bei seiner Erfüllung aber die Qualität des Planes. Dies kann erreicht werden, indem realisierte Wünsche sich positiv auf die Kostenfunktion auswirken und somit die Qualität des Planes steigern.

3.2 Zusammenfassung der Anforderungen

Die Abbildung der Anforderungen in der Wissensbasis sollte so flexibel wie möglich durchgeführt werden können. Dazu ist die freie Definition von Tagen, Räumen, Zeiten, Veranstaltungen und Teilnehmern notwendig.

Weiter muss die Reihenfolge der Zeiten festgelegt werden, denn nur so kann die Forderung nach Mehrfachstunden für einzelne Veranstaltungen durchgesetzt werden.

[1] Im System definierte Mehrfachstunden müssen zwangsläufig als solche umgesetzt werden. Ist es nicht mögliche, einen Plan auf Grund dieser Anforderungen zu erstellen, kann das Programm diese nicht bedarfsweise auflösen, wie es ein menschlicher Stundenplaner könnte. Dem Programm fehlt das Wissen, welche harten Einschränkungen aufgeweicht werden können.

[2] Besser bedeutet in diesem Falle breitere Akzeptanz bei den Teilnehmern.

Die harten Einschränkungen für die Veranstaltungen und die Teilnehmer müssen auf jeden Fall berücksichtigt werden.

Wünsche und harte Einschränkungen sollten aus Gründen der Arbeitserleichterung über dieselben Namen angesprochen werden. Wünsche müssen dabei zusätzlich eine Priorität erhalten, so dass das System sie unterschiedlich gewichten kann. Anders als harte Einschränkungen dürfen Wünsche sich gegenseitig ausschließen und den harten Einschränkungen widersprechen. Widersprüche führen dann dazu, dass der Wunsch nicht berücksichtigt wird. Dies sollte sich nicht negativ auf die Kostenfunktion und den daraus resultierenden Plan auswirken.

Der Arbeitserleichterung dienen auch die fehlende Unterscheidung zwischen Dozenten und Assistenten und die Gleichbehandlung von drei der vier Veranstaltungsarten. Termine nach Absprache werden nicht berücksichtigt, da für sie kein Termin gefunden werden muss.

4 Systementwurf

Ein Stundenplan muss das Problem lösen, alle definierten Vorlesungseinheiten[1]auf die freien Zeiteinheiten der Tage und Räume des Planes zu verteilen. Somit wird eine Veranstaltung durch ein 3-Tupel `(Raum, Tag, Zeit)` klassifiziert.

Bei der Platzierung dürfen keine harten Einschränkungen verletzt werden. Deshalb werden die harten Einschränkungen auf die einzelnen Veranstaltungen umgerechnet und diesen als Constraints zugewiesen. So wird der Suchraum schon vor der eigentlichen Suche eingeschränkt. Außerdem kann durch die Belegung der Veranstaltungen mit Einschränkungen während des Suchablaufs darauf verzichtet werden, die vorgeschlagene Vorlesungsplatzierung zu überprüfen.

Es gibt zwei Arten von unterschiedlichen Einschränkungen: essentielle Einschränkungen, die vom Stundenplanersteller nicht beinflussbar sind, und harte Einschränkungen, die veränderbar sind. Essentielle Einschränkungen bedeuten,

- dass in einem Raum zu einer Zeiteinheit nur eine Veranstaltung stattfinden kann,
- dass eine Veranstaltung nur in einem Raum stattfinden kann, der den Anforderungen entspricht und
- dass ein Teilnehmer zu einer Zeiteinheit nur an einer Veranstaltung teilnehmen kann.

Harten Einschränkungen sind

- die Formulierung der Verfügbarkeit eines Teilnehmers,

[1] Anders als in [dip99] wird nicht auf die Vorlesungen insgesamt, sondern auf einzelne Veranstaltungen innerhlan der Vorlesung zugegriffen.

- die Einschränkung einer Vorlesung auf einen oder eine bestimmte Art von Raum,
- die Definition von Mehrfachstunden und
- der räumliche und zeitliche Zusammenhang der Mehrfachstunden.

In der verfügbaren Literatur werden diese beiden Einschränkungsarten nicht unterschieden. Die Unterscheidung erscheint aber sinnvoll, da nur dadurch die Untergrenze für einen allgemein gültigen Plan definiert werden kann. Essentielle Einschränkungen gelten für jeden Stundenplan, harte Einschränkungen beschreiben den Stundenplan der Fachhochschule Wedel/PTL näher. Werden schon die essentiellen Einschränkungen verletzt, kann kein gültiger Stundenplan mehr erstellt werden.

Für einen schnellen Zugriff auf die einzelnen Veranstaltungen, Tage, Zeiten und Räume werden alle Eingaben indiziert. Für die essentiellen und die harten Einschränkungen werden Umrechnungen auf die einzelnen Veranstaltungen vollzogen, so dass die Zugriffe schnell und gezielt sein können.

Wenn eine Veranstaltung gleichzeitig zwei (oder mehr) Räume benötigt, muss eine zweite, fiktive Veranstaltung in die Berechnung eingezogen werden.

Damit nicht bei jedem Schritt eine Liste mit allen Veranstaltungen durchlaufen werden muss, wird ein Array eingeführt, das alle Veranstaltungen aufnehmen kann.

Dieses Array ist die Schnittstelle, über die die einzelnen Teile der Anwendung miteinander kommunizieren.

Im Folgenden soll näher auf die Berechnung der Constraints, die den Lösungsraum anhand der gegebenen Einschränkungen reduzieren, eingegangen werden. Die Berechnung wird direkt an die einzelnen Elemente des Arrays gekoppelt. Die Auswertung der Einschränkungen kann komplett `ECLiPSe` überlassen werden. Ein gültiger Plan entsteht nur, wenn alle Einschränkungen aufgelöst werden können.

4.1 Die Erstellung der essentiellen Einschränkungen

4.1.1 Die Einschränkungen bezüglich der Räume

Veranstaltungen dürfen nur in Räumen stattfinden, die ausreichenden Platz bieten und die nötigen Ressourcen bereitstellen. Dazu wird die Raummenge anhand der Wissensbasis auf die belegbaren Räume begrenzt.
Bei der Lösungsfindung wird nur noch auf diese eingeschränkte Raummenge zugegriffen.

- **Raumkollsionen**

  Um zu verhindern, dass in einem Raum gleichzeitig zwei Veranstaltungen stattfinden, wird über alle Elemente des Arrays die folgende Einschränkung angegeben:

  $$\begin{aligned} &ZahlTage * ZahlZeiten * AktuellerRaum \\ &\quad + ZahlTage * AktuellerTag + AktuelleZeit = X \end{aligned} \tag{4.1}$$

  Dabei stellt $ZahlTage$ die Zahl aller Tage pro Woche und *ZahlZeiten* die Zahl aller Zeiteinheiten pro Tag dar. Bei *AktuellerRaum*, $AktuellerTag$ und $AktuelleZeit$ handelt es sich um die Indizes der zugehörigen Räume, Tage und Zeiten. Die so erstellten Einschränkungen werden in einer Liste gesammelt. Die Eindeutigkeit der einzelnen Elemente X muss durch das PROLOG-Prädikat `alldifferent/1` sichergestellt werden.

4.1.2 Die Einschränkungen bezüglich der Teilnehmer

Für alle Teilnehmergruppen wird eine Liste von Veranstaltungen erstellt, die nicht zur selben Zeit stattfinden dürfen. Mit Hilfe dieser Liste wird erreicht, dass ein Teilnehmer während einer Zeiteinheit nur an einer Veranstaltung teilnehmen kann.

- **Teilnehmerkollisionen**

  Teilnehmerkollisionen werden ähnlich wie Raumkollisionen vermieden.

  Dazu wird jedem Listenelement der Teilnehmergruppenliste die folgende Einschränkung zugewiesen:

  $$ZahlZeiten * AktuellerTag + AktuelleZeit = X \tag{4.2}$$

  $ZahlZeiten$, $AktuellerTag$ und $AktuelleZeit$ sind wie unter 4.1.1 beschrieben belegt. Auch hier wird die Eindeutigkeit der Listenelemente durch das Aufstellen einer Constraintliste mit Hilfe von `alldifferent/1` sichergestellt.

4.2 Die Erstellung der harten Einschränkungen

4.2.1 Einschränkungen für Veranstaltungen

Im einfachsten Fall ist eine Veranstaltung in ihrem Suchraum nicht eingeschränkt, so dass keine Constraints notwendig sind.
Wahrscheinlicher ist, dass eine Veranstaltung in ihrem Suchraum durch zeitliche und räumliche Vorgaben eingeschränkt ist.

4.2.1.1 Festlegung einer Veranstaltung auf einen bestimmten Tag und/oder Zeit

`AmUmEinschraenkungen`
Diese Einschränkung wird über folgende Constraints erreicht:

$$Tage :: ListeGewaehlterTage \tag{4.3}$$

$$Zeiten :: ListeGewaehlterZeiten \tag{4.4}$$

$ListeGewaehlterTage$ und $ListeGewaehlterZeiten$ enthalten die Tage und Zeiten, die noch für diese Veranstaltung zur Verfügung stehen.

Die gefundenen Einschränkungen werden den schon auf diesem Constraint liegenden Einschränkungen zusätzlich zugewiesen und in die weiteren Berechnungen einbezogen.

4.2.1.2 Sperren einer Veranstaltung zu bestimmten Tagen und/oder Zeiten

`NichtAmUmEinschraenkung`
Diese Einschränkungen werden dem System mit der folgenden Berechnung übergeben:

$$\begin{aligned}(ZahlZeiten * AktuellerTag + AktuelleZeit)\\ \neq (ZahlZeiten * Tag_n + Zeit_n)\end{aligned} \tag{4.5}$$

$ZahlZeiten$, $AktuellerTag$ und $AktuellerZeit$ sind wie unter 4.1.1 beschrieben belegt. Tag_n und $Zeit_n$ stehen für die zu bearbeitende Veranstaltung.
Um das System weniger komplex zu halten, werden `NichtAmUmEinschraenkungen` nur angelegt, wenn sie sich nicht durch `AmUmEinschraenkungen` ersetzen lassen. Dies liegt daran, dass `NichtAmUmEinschraenkungen` mit dieser Berechnungsmethode nicht direkt auf die Lösungsmenge zugreifen, sondern nur durch eine zusätzliche Berechnung im Falle einer Belegung überprüft werden können.

4.2.1.3 Festlegung, dass Veranstaltungen am selben Tag stattfinden müssen

`AmSelbenTagEinschraenkung`
Diese Einschränkung wird mit dem folgenden Constraint umgesetzt:

$$Tag_n = Tag_m \tag{4.6}$$

Tag_m und Tag_n stehen für die Tage, an dem die Veranstaltungen stattfinden sollen.
So wird erreicht, dass zwei Veranstaltungen an einem Tag stattfinden.

4.2.1.4 Festlegung, dass Veranstaltungen nicht am selben Tag stattfinden dürfen

`NichtAmSelbenTagEinschraenkung`

Die `NichtAmSelbenTagEinschraenkung` wird analog zur `AmSelben-TagEinschraenkung` umgesetzt. Diesmal wird jedoch eine Ungleichheit gefordert.

$$Tag_n \neq Tag_m \tag{4.7}$$

Tag_m und Tag_n stehen für die Tage, an denen die Veranstaltungen stattfinden sollen.
Durch die $\neq$-Forderung wird erreicht, dass zwei Veranstaltungen nicht am selben Tag stattfinden.

4.2.1.5 Festlegen, dass Veranstaltungen nicht hintereinander im Plan auftreten dürfen

`NichtZusammenhaengendEinschraenkung`

Die Einschränkung, dass zwei Veranstaltungen nicht hintereinander im Plan auftreten dürfen, wird über die nachfolgende Formel eingegeben:

$$(Zeit_m \neq Zeit_n + 1) \bigwedge (Zeit_m + 1 \neq Zeit_n) \tag{4.8}$$

$Zeit_m$ und $Zeit_n$ steht für die Zeit, zu der eine Veranstaltung stattfinden soll.

4.2.1.6 Festlegung, dass zwei Veranstaltungen zur selben Zeit stattfinden müssen

`ZurGleichenZeitEinschraenkung`

Diese Einschränkung wird genutzt, um durchzusetzen, dass einer Veranstaltung zu einer Zeit zwei Räume zugewiesen werden.
Über die folgenden Einschränkungen werden diese Constraints dem System zugeführt:

$$Tag_m = Tag_n \tag{4.9}$$

$$Zeit_m = Zeit_n \tag{4.10}$$

Tag_m und Tag_n und $Zeit_m$ und $Zeit_n$ stehen für den Tag und die Zeit der jeweiligen Veranstaltungen.

4.2.2 Die Einschränkungen bezüglich der Mehrfachstunden

Bestimmte Veranstaltungen stehen in räumlichem und zeitlichem Zusammenhang und müssen daher auch bei der Planerstellung geblockt werden. Da bereits festgelegt ist, dass räumliche und zeitliche Blockung sich nicht über mehrere Tage erstrecken kann, ergeben sich die folgenden Constraints für die Festsetzung von Doppelstunden:
Für die räumliche Blockung, wird das folgende Constraint eingeführt:

$$Raum_n = Raum_m \tag{4.11}$$

Dass die Mehrfachstunden an einem Tag stattfinden, erreicht das folgende Constraint:

$$Tag_n = Tag_m \tag{4.12}$$

Um die Reihenfolge der Veranstaltungen festzulegen, wird die dritte Einschränkung hinzugenommen:

$$Zeit_n = (Zeit_m + 1), \tag{4.13}$$

$Raum_i$, Tag_j und $Zeit_k$ stehen für die betroffenen Räume, Tage und Zeiten.
Drei- und Vierfachstunden lassen sich auf Doppelstunden umrechnen, so dass

```
dreifachstunde('Physikalisches Pr (B) 1', 'Physikalisches
Pr (B) 2', 'Physikalisches Pr (B) 3')
```

auf

```
doppelstunde('Physikalisches Pr (B) 1', 'Physikalisches Pr
(B) 2'),
doppelstunde('Physikalisches Pr (B) 2', 'Physikalisches Pr
(B) 3')
```

abgebildet wird.

4.2.3 Die Einschränkungen bezüglich der Teilnehmer

Die Einschränkungen der Teilnehmer werden im System auf die einzelnen Veranstaltungen umgerechnet. Danach werden die Einschränkungen mit Hilfe der schon bekannten Berechnungen im System durchgesetzt. Weitere Berechnungen für die Teilnehmer sind nicht notwendig.

5 Umsetzung

In diesem Kapitel geht es um die technische Umsetzung und die Programmarchitektur.

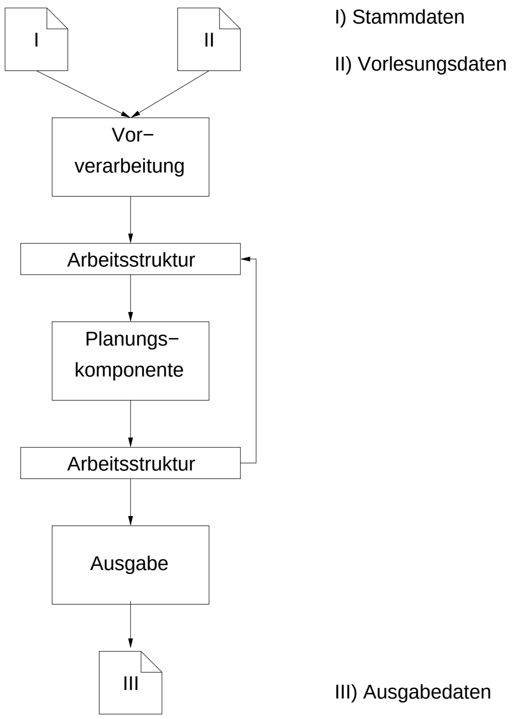

Abbildung 5.1: Allgemeine Übersicht des Programmablaufs

Aus den Anforderungen und der Entwurfsbeschreibung lassen sich drei Arbeitsphasen ableiten:

- **Die erste Phase - Vorverarbeitung**
  In der ersten Phase werden die Constraintvariablen aufgebaut und die Constraints über diese Variablen erstellt.

- **Die zweite Phase - Planungskomponente**
  In der zweiten Phase werden die Constraintvariablen mit Werten aus ihrem Wertebereich belegt.
- **Die dritte Phase - Ausgabe**
  Die dritte Phase gibt das Ergebnis der Lösungssuche aus.

Diese drei Phasen entsprechen den unter 2.4.2 vorgestellten Phasen eines Constraint-Programmes.
Abbildung 5.1 zeigt diese drei Phasen. Das System wird durch zwei Prologdateien, die `Stammdaten` und die `Vorlesungsdaten`, gespeist. In der ersten Phase, der Vorverarbeitung, wird die Schnittstelle erstellt und die Veranstaltungsplatzierung vorbereitet. In der zweiten Phase, der eigentlichen Planung, wird der Suchraum jeder Veranstaltung so weit reduziert, bis die Veranstaltung korrekt im Plan platziert ist. In der dritten Phase, der Ausgabe, wird der erzeugte Plan in Form von Prologfakten ausgegeben.

5.1 Die Vorverarbeitung

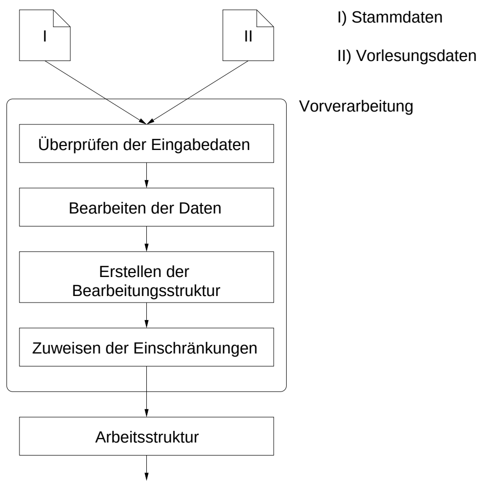

Abbildung 5.2: Übersicht der Vorverarbeitung

Die Vorverarbeitung nutzt die unter 4 genannten Formeln, um die Einschränkungen in das System einzubeziehen. Dazu wird ein allgemeines

Array mit dem Wertebereich `1 .. AnzVeranstaltungen` erstellt. Die Constraints werden während der Vorverarbeitung dem Array zugewiesen.

5.1.1 Überprüfen der Eingabedaten

In diesem Abschnitt wird durch einen einfachen Test die Wissensbasis (`Stammdaten` und `Vorlesungsdaten`) überprüft, um einfache Fehler zu finden. Dieser Test kann allerdings nicht Fehler in den komplexeren Anforderungen und in den Abhängigkeiten von harten Constraints entdecken.

5.1.2 Bearbeiten der Daten

Im nächsten Schritt werden die `Stammdaten` und die `Vorlesungsdaten` aufbereitet und indiziert. Fortan werden nur noch diese indizierten Daten benutzt.

5.1.3 Erstellen der Bearbeitungsstruktur

Dieser Schritt erstellt dynamisch das Anfangsarray. Außer der Anzahl der Elemente hat das Array zunächst keinen weiteren Bezug zum System.

```
einStundenplan(stundenplan(ver(Raum1, Tag1, Zeit1), ...,
ver(RaumN, TagN, ZeitN)))
```

5.1.4 Zuweisen der Einschränkungen

Erst dadurch, dass die Einschränkungen dem Array zugewiesen werden, wird es mit dem System verbunden, so dass der Suchraum immer weiter eingeschränkt werden kann, um unnötige Berechnungen zu vermeiden.

1. Danach werden den einzelnen Arrayelementen alle Wertebereiche für `Raum :: 1 .. ZahlRaeume`, `Tag :: 1 .. ZahlTage` und `Zeit :: 1 .. ZahlZeiten` zugewiesen. Diese Wertebereiche werden dabei nicht eingeschränkt. `Raum`, `Tag` und `Zeit` werden in `ECLiPSe` auch als `Domain` bezeichnet.

```
stundenplan(ver(Raum1[1..23], Tag1[1..6],
Zeit1[1..8]), ...)
```

2. Dann werden die essentiellen und die harten Einschränkungen einbezogen. Im Lösungsraum direkt erkennbar werden nur die Einschränkungen, die diesen sichtbar begrenzen. Die anderen Constraints sind als Berechnungen an die einzelnen Arrayelemente gebunden, während die eigentlichen Berechnungen bei der Planerstellung erfolgen.

   Einschränkungen, die direkt im Wertebereich sichtbar sind, sind leicht zu erkennen.

   ```
   stundenplan(ver(2, Tag2[2..6], Zeit1[1..7]), ...)
   ```

   Die anderen Einschränkungen werden im System als Berechnungen präsentiert:

$$96 + 8 * Tag2[2..5] + Zeit1[1..6] - Kopf[113..142] = 0 \tag{5.1}$$

$$0 + 8 * Tag2[2..5] + Zeit1[1..6] - Kopf[17..46] = 0 \tag{5.2}$$

   Dies sind Berechnungen zur Umsetzung der Raumkollisionen aus Abschnitt 4.1.1 für die erste Veranstaltung zu zwei aufeinanderfolgenden Zeitpunkten. Die erste und die zweite im Array gespeicherte Veranstaltung müssen demnach am selben Tag stattfinden. `ECLiPSe` verwaltet die Einschränkungen gemeinsam, die zusammengeführt werden können.

   Dieser Schritt führt zu der in der Abbildung 5.2 gezeigten Arbeitsstruktur.

5.2 Die Planungskomponente

In dem System gibt es keinen festen Planungsalgorithmus. Vielmehr kann die Planungskomponente beliebig ausgetauscht werden. Die Planungskomponente beeinflusst, wie Abbildung 5.3 zeigt, die Arbeitsstruktur. Nach der Planung wird die gelöste Arbeitsstruktur an die Ausgabe gegeben.

Die Planungskomponente muss somit Mengen wie `Tag2[2..5]` immer weiter reduzieren können, bis nur noch ein einziges Element in der Menge vorhanden ist, welches als Teil des Stundenplans übernommen werden kann.

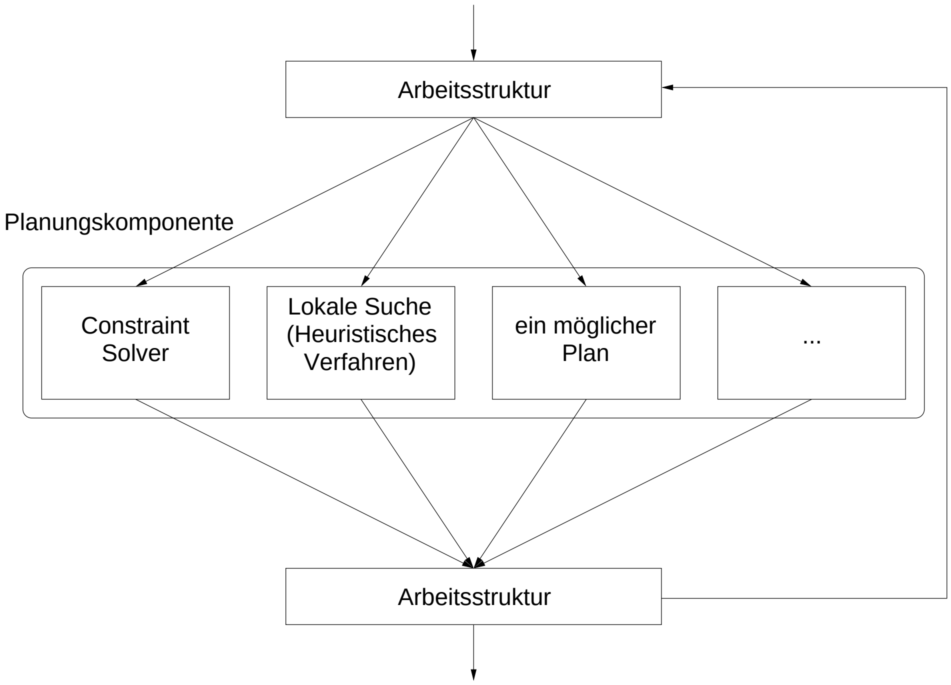

Abbildung 5.3: Übersicht über die Planungskomponente

Die einfachste Planungskomponente hat die folgende Struktur:

```
verarbeitung(Stundenplan) :-
       anzVeranst(AnzVeranst),
    labeling(Stundenplan, AnzVeranst).

labeling(_, 0).
labeling(Stundenplan, Index) :-
    subscript(Stundenplan, [Index, 1], Raum),
    subscript(Stundenplan, [Index, 2], Tag),
    subscript(Stundenplan, [Index, 3], Zeit),
    indomain(Raum),
    indomain(Tag),
    indomain(Zeit),
    IndexNeu is Index - 1,
    labeling(Stundenplan, IndexNeu).
```

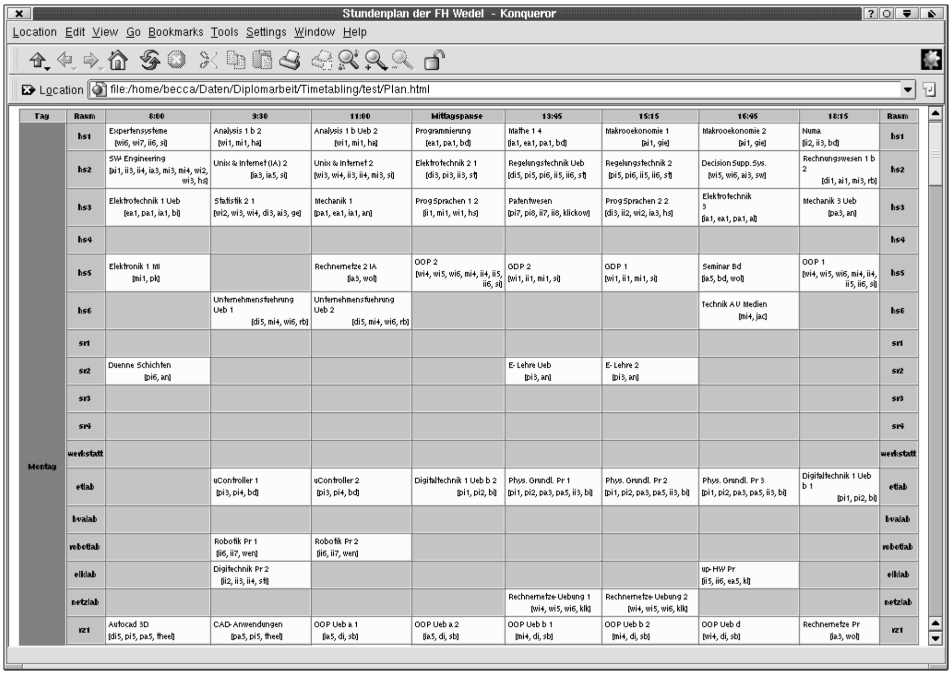

Abbildung 5.4: Mit der einfachsten Planungskomponente auf Grundlage der Wissensbasis vom Wintersemester 1998/99 erstellter Plan

Mit `verarbeitung(Stundenplan)` wird die Planungskomponente gestartet. Das Array wird während des Suchablaufs, wie gefordert, reduziert. Dies geschieht über `labeling(Stundenplan, Index)`. `labeling` bezieht die Standardsuchmethode von `ECLiPSe` ein. Für die Planungskomponente muss diese Standardsuchmethode angepasst werden, da sie nur eine einfache Liste, deren Elemente auf einer `Domain` basieren, verarbeiten kann. Im vorliegenden Fall gibt es jedoch drei Wertebereiche. Das `labeling`-Prädikat ist in `ECLiPSe` folgendermaßen umgesetzt[1]:

```
labeling([]).
labeling([Kopf|Rest]) :-
    indomain(Kopf),
    labeling(Rest).
```

[1]Entnommen aus der Dokumentation zu `ECLiPSe`

`indomain` instanziiert eine Variable mit dem kleinsten Wert ihres Wertebereiches.

Der mit dem neuen `labeling`-Prädikat erstellte Plan erfüllt alle an ihn gestellten Anforderungen, obwohl er so einfach ist. Mit etwas Aufwand kann eine bessere Veranstaltungsplatzierung erreicht werden, indem die einfache Lösung erweitert wird.

Sollen Wünsche bei der Platzierung berücksichtigt werden, müssen diese in der Planungskomponente ausgewertet werden. Dazu können Berechnungen ähnlich denen bei der Umsetzung der harten Constraints genutzt werden.

5.3 Die Ausgabe

Im letzten Schritt werden die Ergebnisse in eine lesbare Form gebracht. Dazu wird das vollständig gelöste Array, das einen korrekten Plan enthält, durch die Auflösung der Indizes wieder in Veranstaltungen, Räume, Tage und Zeiten umgewandelt. Die Ausgabedatei entspricht danach im Format der Eingabedatei. In einem erneuten Durchlauf könnte die Ausgabedatei als Eingabedatei genutzt werden, um z.B. die Raumbelegung zu ändern.

Die gefundenen Platzierungen werden nach Tagen, Räumen und Zeiten sortiert in die Ausgabedatei geschrieben, da so die weitere Verarbeitung der Ergebnisse in anderen Sprachen einfacher ist.

5.4 Die Aufteilung des Codes

Der Code des entwickelten Systems ist auf folgende Dateien verteilt:

- `stundenplanGenerierung.pl`
  In dieser Datei wird der Ablauf der Planerstellung gesteuert.
  - `allgemeineFunktionalitaeten.pl`
    Diese Datei enthält allgemeine Funktionalitäten, die in den unterschiedlichen Dateien immer wieder benötigt werden.
  - `tests.pl`
    In dieser Datei wird die Wissensbasis z.B. auf Tippfehler überprüft und es werden einige weitere, wenig komplexe Kontrollen der Wissensbasis vorgenommen.

- `vorverarbeitung.pl`
  In dieser Datei wird der Aufbau der Schnittstelle, ein Array, eingeleitet.
  - `essentielleEinschraenkungen.pl`
    Hier wird dafür gesorgt, dass ein Teilnehmer zu einer bestimmten Zeiteinheit nicht an zwei Veranstaltungen gleichzeitig teilnehmen muss, und dass in einem Raum zu einer Zeiteinheit immer nur eine Veranstaltung stattfindet.
  - `harteConstraints.pl`
    In dieser Datei werden die harten Einschränkungen aus der Wissensbasis auf das Array übertragen.
- `schreibePlan.pl`
  Nach der Berechnung wird der fertige Plan in eine Ausgabedatei geschrieben.

Die beschriebenen Dateien bilden die Grundstruktur des Systems, das durch eine Ausgaberoutine ergänzt wird, die den fertigen Plan zunächst in `xml` und dann mit Hilfe von `xsl` in das `HTML`-Format umwandelt.

Diese Ausgaberoutine dient zunächst der Kontrolle des Plans. Entweder können alle Teilnehmer oder nur eine bzw. mehrere Teilnehmergruppen ausgewählt werden. In diesem Fall werden nur Veranstaltungen angezeigt, an denen mindestens eine der ausgewählten Teilnehmergruppen teilnimmt. Alle Raumbelegungen werden farblich hervorgehoben, damit Verbesserungsmöglichkeiten leichter zu erkennen sind.

5.5 Interne Datenstrukturen

Die internen Datenstrukturen enthalten im Wesentlichen das Array, das während des Programmablaufes genutzt wird. Es befindet sich während des gesamten Programmablaufs nur im Speicher.

Außerdem gehören zu den internen Datenstrukturen die indizierten Angaben, die aus der Wissensbasis stammen. Die Umrechnungen werden in der Datei `Ergebnis.pl` als Prologfakten zwischengespeichert. Damit stehen sie immer zur Verfügung und können z.B. zur Kontrolle eingesehen werden. Außerdem wird so der Einsatz von `retract/1`

und `assert/1` (und verwandten Prädikaten) vermieden, was sich positiv auf die Lesbarkeit und Wartbarkeit des Codes auswirkt.
Die Datei `Ergebnis.pl` enthält die internen Datenstrukturen in der folgenden Reihenfolge:

1. Essentielle Einschränkungen
   a) Indizierung der Tage, Zeiten und Räume
   b) Anzahl der Tage, Zeiten und Räume
   c) Zusammenfassung aller Teilnehmer
   d) Indizierung aller Teilnehmer (ohne Berücksichtigung der Untergruppen)
   e) Indizierung der Veranstaltungen
   f) Anzahl der Veranstaltungen
2. Grundstruktur des Arrays
3. Harte Einschränkungen und Wünsche, jeweils in folgender Reihenfolge:
   a) Indizierung der Veranstaltungen, die nicht parallel stattfinden dürfen
   b) Indizierung der Veranstaltungen, die parallel stattfinden müssen
   c) Indizierung des Ergebnisses der Umrechnungen aller Mehrfachstunden auf Doppelstunden
   d) Indizierung der Veranstaltungen, die nicht am selben Tag stattfinden dürfen
   e) Indizierung der Veranstaltungseinschränkungen, die festlegen, dass Veranstaltungen zu einem bestimmten Zeitpunkt stattfinden müssen
   f) Indizierung der Veranstaltungseinschränkungen, die bestimmte Termine aus der Platzierungsmenge der Veranstaltungen entfernen
   g) Ergebnis der Umrechnungen der Teilnehmereinschränkungen

h) Indizierung des Ergebnisses der Umrechnungen der Teilnehmereinschränkungen

i) Indizierung der Veranstaltungen, die nicht hintereinander im Plan auftreten dürfen

j) Indizierung der Veranstaltungen, die am selben Tag stattfinden müssen

Zur einfachen Unterscheidung werden dabei harte Einschränkungen durch `Faktname + V` und Wünsche durch `Faktname + W` repräsentiert.

5.6 Zusammenfassung

Der Ablauf des Gesamtsystems ist wie folgt:

1. Start des Systems durch den Aufruf des Prädikates `start/2` aus `stundenplanGenerierung.pl`.

   a) Kompilieren der allgemeinen Funktionalitäten aus `allgemeineFunktionalitaeten.pl`.

   b) Kompilieren der angegebenen Stamm- und Vorlesungsdaten.

2. Starten der Vorverarbeitung mit Hilfe des Prädikates `vorverarbeitung/1` aus der Datei `vorverarbeitung.pl`.

   a) Überprüfen der angegebenen Wissensbasis mit Hilfe des Prädikates `fuehreTestDurch/0` aus `tests.pl`.
   Bei einem Fehler wird der Programmlauf mit einer entsprechenden Fehlermeldung abgebrochen. Hinweise zur Fehlerbehebung finden sich im Anhang B.

   b) Bearbeiten der Wissensbasis mit den Prädikaten `bearbeiteStammdaten/1` und `bearbeiteVorlesungsdaten/1` und erstellen der Bearbeitungsstruktur mit Hilfe des Prädikates `erstelleArray/1` aus der Datei `vorverarbeitung.pl`.

   c) Aufbau der internen Datenstrukuren, die aus der Wissensbasis abgeleitet werden können.

d) Zuweisen der essentiellen und der harten Einschränkungen unter Verwendung der Prädikate `berechneEssentielle-Einschraenkungen/1` und `berechneHarteConstraints/1` aus den Dateien `essentielleEinschraenkungen.pl` und `harteConstraints.pl`.

3. Starten der Planungskomponente über den Aufruf von `verar-beitung/1` in der zugehörigen Datei.

4. Ausgabe des Ergebnisses in Form von PROLOG-Fakten über den Aufruf `schreibePlan/1` in der Datei `schreibePlan.pl`

5. Verarbeitung in eine einfach lesbare Darstellung des Plans, z.B. in `HTML`.

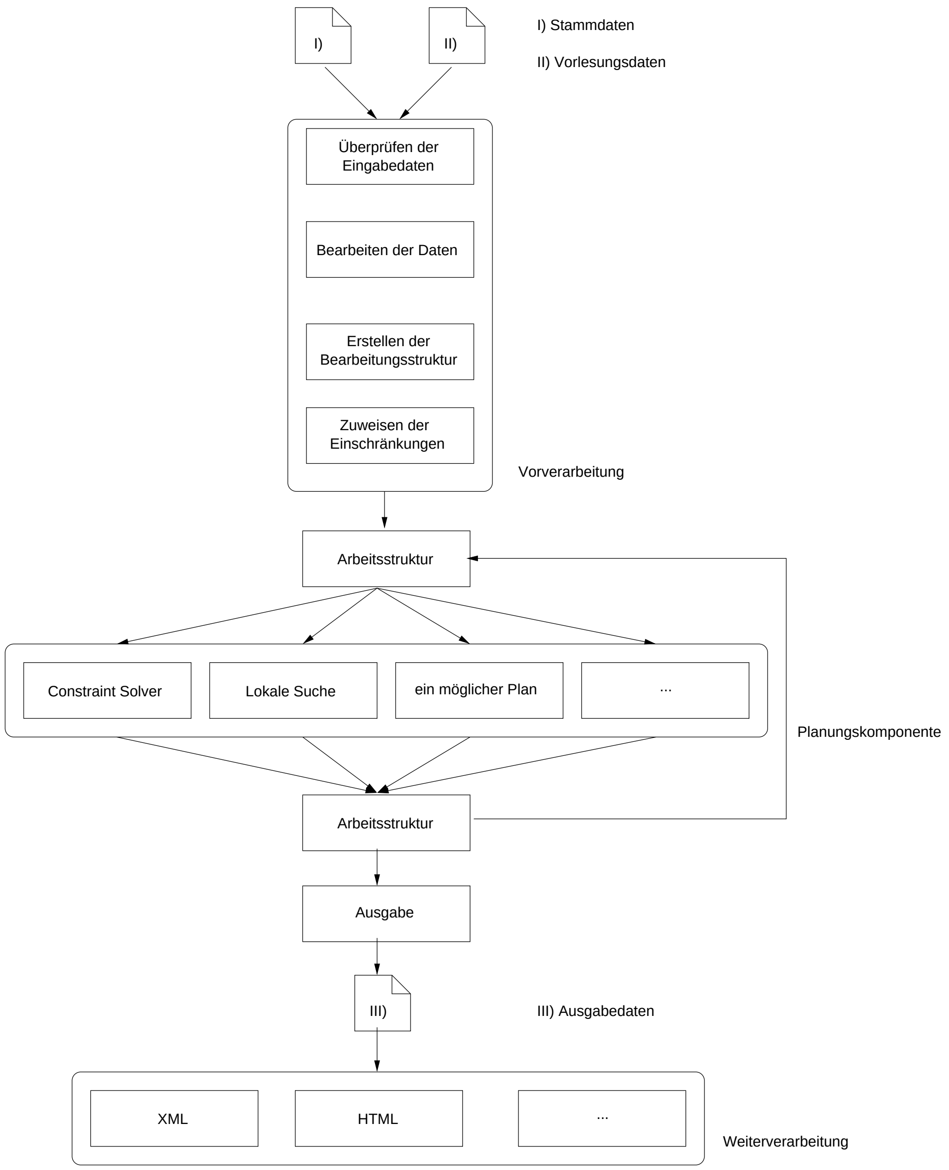

Abbildung 5.5: Übersicht über den Gesamtablauf des Systems

6 Eine komplexe Planungskomponente - die konstruktive Suche

Der vorgestellte Algorithmus orientiert sich an der Vorgehensweise eines Stundenplanmachers bei der Platzierung von Veranstaltungen in einem Stundenplan.
Bei dem verwendeten Algorithmus handelt es sich um eine Tiefensuche mit automatisch ausgelöstem Backtracking (constructive search).
Die Vorgehensweise eines Planerstellers nach der Eingabe der Vordaten kann vereinfacht so beschrieben werden:

1. Festlegen der Platzierungsreihenfolge.
2. Vermeiden ungünstiger Tage und Zeiten.
3. Durchsetzen eines ausgewogenen Plans mit einer gleichmäßigen Auslastung der Räume, Tage und Zeiten.

Im Folgenden soll dieser Ablauf in PROLOG umgesetzt werden.

6.1 Die Anforderungen

Jede Fehlplatzierung einer Veranstaltung kann dazu führen, dass der Algorithmus keine Lösung in angemessener Zeit findet. Ein gegen Ende des Platzierungsvorganges ausgelöstes Backtracking kann zu einem zu hohen Zeitaufwand führen.

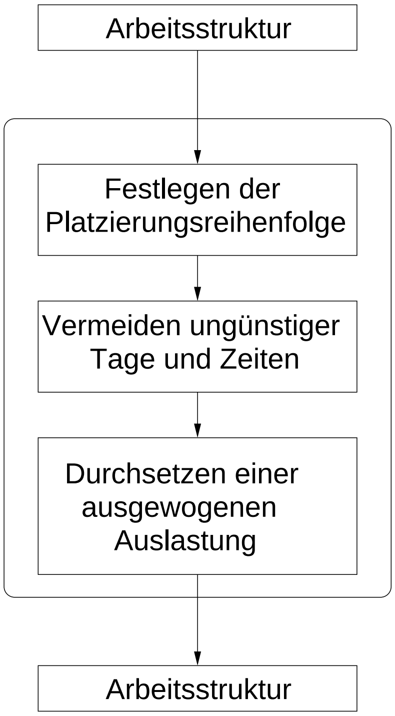

Abbildung 6.1: Übersicht über die einzelnen Schritte der Planungskomponente

6.1.1 Die Platzierungsreihenfolge

Die Veranstaltungen, die von vornherein auf einen Tag, eine Zeit und einen Raum festgelegt sind, müssen zuerst platziert werden. Danach müssen Veranstaltungen, bei denen ein Teilnehmer in seinem zur Verfügung stehenden Zeitraum stark eingeschränkt ist, geplant werden. Auch Mehrfachstunden sind als zeitlicher und räumlicher Block schwer zu platzieren. Der Platzierungsspielraum für solche Veranstaltungen ist von vornherein stark eingeschränkt.

Des Weiteren sind Veranstaltungen mit einer hohen Zahl unterschiedlicher Teilnehmergruppen ebenfalls schwierig im Plan unterzubringen, weil Kollisionen von Teilnehmergruppen vermieden werden müssen.

Erst wenn alle „schwierigen“ Veranstaltungen gesetzt sind, werden alle Veranstaltungen mit wenigen oder ohne Einschränkungen berücksichtigt.

6.1.2 Die Vermeidung ungünstiger Tage und Zeiten

Ungünstige Tage und Zeiten sollten bei der Planung nur im Notfall benutzt werden. Ein Plan verliert stark an Qualität, wenn viele Veranstaltungen in den späten Abendstunden und/oder am Sonnabend liegen und würde wohl von allen Beteiligten heftig kritisiert werden.

Eine Veranstaltung sollte nur in die Abendstunden und auf den Sonnabend gelegt werden, wenn dies ausdrücklich gefordert wird oder wenn durch die Einschränkungen ein anderer Termin unmöglich ist.

6.1.3 Die ausgewogene Auslastung aller Ressourcen

Eine gültige und die essentiellen Einschränkungen berücksichtigende Platzierung der Veranstaltungen unter Beachtung aller harten Constraints soll auch zu einer gleichmäßigen Auslastung der Ressourcen führen. Alle Veranstaltungen sollten über Räume, Tage und Zeiten ausgewogen verteilt werden.

6.2 Die Umsetzung

Um die Anforderungen umzusetzen, müssen unterschiedliche Wissensbasen verarbeitet werden können, wobei feste Vorgaben möglichst vermieden werden sollten.

Unter Umständen sind mehrere Läufe notwendig, bis eine funktionierende Reihenfolge der Veranstaltungen gefunden wird.

6.2.1 Die Platzierungsreihenfolge

Das Programm muss in der Lage sein, stark eingeschränkte Veranstaltungen in der Wissensbasis zu finden und an den Anfang des Platzierungsablaufs zu setzen.

Bei der Umsetzung in PROLOG muss sichergestellt sein, dass trotz fester Vorgaben genügend Flexibilität erhalten bleibt.

Dazu kann der Planersteller über eine lokale Wissensbasis, die Datei `WBHeuristik.pl`, die Platzierungsreihenfolge beeinflussen, ohne in den Sourcecode einzugreifen.

6.2.1.1 Feste Veranstaltungen

In dieser Planungskomponente werden feste Veranstaltungen mit Hilfe des Prädikates `delayed_goals_number/2` umgesetzt. `delayed_goals_number/2` berechnet die auf einer Variablen liegenden Einschränkungen. Der Rückgabewert gibt die Anzahl der Einschränkungen an. Ist einer Variablen bereits ein Wert fest zugewiesen, liefert `delayed_goals_number/2` den Wert 1.000.000 zurück. Bei einer Veranstaltung, die auf `Raum`, `Tag` und `Zeit` festgelegt ist, wird bei der Berechnung der Wert 3.000.000 zurückgegeben.

```
berechneFesteVeranstaltungen(_, 0, []).
berechneFesteVeranstaltungen(Stundenplan, Index,
                [Kopf|Rest]) :-
    subscript(Stundenplan, [Index, 1], Raum),
    subscript(Stundenplan, [Index, 2], Tag),
    subscript(Stundenplan, [Index, 3], Zeit),
    delayed_goals_number(Raum, RaumC),
    delayed_goals_number(Tag, TagC),
    delayed_goals_number(Zeit, ZeitC),
    Ergebnis is (RaumC + TagC + ZeitC),
    Ergebnis >= 3000000,
    append([Ergebnis], [Index], Kopf),
    IndexNeu is Index - 1,
    berechneFesteVeranstaltungen(Stundenplan, IndexNeu,
                    Rest).
berechneFesteVeranstaltungen(Stundenplan, Index, Ergebnis) :-
    IndexNeu is Index - 1,
    berechneFesteVeranstaltungen(Stundenplan, IndexNeu,
                    Ergebnis).
```

6.2.1.2 Stark eingeschränkte Teilnehmerverfügbarkeiten

In einer komplexen Berechnung wird ermittelt, welche Teilnehmergruppen verfügbar sind. Dabei werden zuerst alle Teilnehmer herausgesucht, bei denen weitergehende Einschränkungen über `teilnehmerNichtAmUmEinschraenkungen` definiert worden sind. Diese Einschränkungen werden auf Zeiteinheiten umgerechnet und mit den Veranstaltungen verglichen, die für diese Teilnehmer verpflichtend sind. Ergibt sich ein sehr geringer Platzierungsspielraum für diese Veranstaltungen, werden diese genauso wie die fest platzierten Veranstaltungen behandelt und an den Anfang der Platzierungsreihenfolge gesetzt.

```
berechneTeilnehmerVerfuegbarkeiten(Ergebnis) :-
    berechneVerfuegbarkeitEinesTeilnehmers(Liste),
    schreibeRelevanteVeranstaltungen(Liste, 3000000, Ergebnis).

berechneVerfuegbarkeitEinesTeilnehmers(Ergebnis) :-
    % Verwaltung
    anzTage(AnzTage),
    anzZeiten(AnzZeiten),
    findall(Tage, tag(Tage, _), TagListe),
    % Intelligenz
    % alle benutzten Praedikate finden sich in
    % 'allgemeineFunktionaltitaeten.pl'
    findall(Teilnehmer, teilnehmerNichtAmUm(Teilnehmer, _, _),
        TeilnehmerListe),
    sort(TeilnehmerListe, TeilnehmerListeSort),
    findeMehrfachAufgefuehrte(TeilnehmerListeSort,
                 MehrfachListe),
    erweitereListeUmTage(MehrfachListe, TagListe,
                NeueTagListe),
    schreibeTage(MehrfachListe, NeueTagListe,
            NeueMehrfachListe),
    flatten(NeueMehrfachListe, NeueMehrfachListeFlat),
    findall((Teilnehmer, Am, Um),
        (teilnehmerNichtAmUm(Teilnehmer, Am, Um),
         nonmember((Teilnehmer, _, _),
               NeueMehrfachListeFlat)),
        NeueTeilnehmerListe),
    append(NeueTeilnehmerListe, NeueMehrfachListeFlat,
          VollstaendigeTeilnehmerListe),
    sort(VollstaendigeTeilnehmerListe,
        TeilnehmerListeSort),
    berechneZurVerfuegungStehendeZeiteinheiten(TeilnehmerListeSort,
                AnzTage, AnzZeiten, ZeiteinheitenListe),
    flatten(ZeiteinheitenListe, ZeiteinheitenListeFlat),
    fuehreErgebnisseZusammen(ZeiteinheitenListeFlat,
                ZeiteinheitenListeKomprimiert),
    flatten(ZeiteinheitenListeKomprimiert,
        ZeiteinheitenListeKomprimiertFlat),
    AnzZeitenUndTage is AnzTage * AnzZeiten,
    % Verwaltung
    passeZahlenAn(ZeiteinheitenListeKomprimiertFlat,
             AnzZeitenUndTage, AngepassteListe),
    findeAnzahlDerVeranstaltungen(AngepassteListe,
                    SpielraumListe),
    rechneAufVeranstaltungenUm(SpielraumListe, ErgebnisListe),
    flatten(ErgebnisListe, Ergebnis).
```

6.2.1.3 Große Teilnehmergruppenanzahl

Eine große Teilnehmergruppenanzahl[1] ist wegen der vielen zu berücksichtigenden Abhängigkeiten schwer zu platzieren und muss deshalb am Anfang verplant werden. Deshalb ist es wichtig, die Platzierung dieser Veranstaltungen nicht zu lange aufzuschieben, da nur zu Beginn der Planung Zeiteinheiten vorhanden sind, die alle Einschränkungen erfüllen.

Um die Veranstaltungen mit großen Teilnehmergruppenanzahlen zu finden, werden in der Planungskomponente einfach die Teilnehmergruppen mit Hilfe von `length/2` gezählt. Übersteigt das Ergebnis eine festgelegte Größe, werden die betroffenen Veranstaltungen als schwer platzierbar eingestuft und mit einer höheren Platzierungspriorität versehen.

```
berechneGrosseTeilnehmerGruppen([], []).
berechneGrosseTeilnehmerGruppen([(ID, Teilnehmer)|Rest],
                    [ErgebnisKopf|ErgebnisRest]) :-
        length(Teilnehmer, N),
        N > 6,
        append([2000000, ID], [], ErgebnisKopf),
        berechneGrosseTeilnehmerGruppen(Rest, ErgebnisRest).
berechneGrosseTeilnehmerGruppen([(_, _)|Rest], Ergebnis) :-
        berechneGrosseTeilnehmerGruppen(Rest, Ergebnis).
```

6.2.1.4 Mehrfachstunden

Wegen der vielen Abhängigkeiten müssen auch Mehrfachstunden bei der Platzierung gesondert behandelt werden. Für Mehrfachstunden müssen im Plan wenigstens zwei Zeiteinheiten hintereinander in einem Raum frei sein.

Dies geschieht in der Planungskomponente so:

```
berechneDreiUndVierfachstunden(Ergebnis) :-
        findeDreifachstunden(DreifachstundenListe),
        findeVierfachstunden(VierfachstundenListe),
        append(DreifachstundenListe, VierfachstundenListe,
            Liste),
        wandleStundenInZahlen(Liste, ZahlenListe),
        flatten(ZahlenListe, ZahlenListeFlat),
```

[1] Bei Teilnehmergruppenanzahl handelt es sich um wirkliche Teilnehmergruppen wie `wi3` und `ii5` und nicht um die Größe einer Teilnehmergruppe, die z.B. zur Bestimmung eines Raumes benötigt wird.

```
        schreibeRelevanteVeranstaltungen(ZahlenListeFlat,
                        2000000, Ergebnis).

berechneDoppelstunden(Ergebnis) :-
        findeDoppelstunden(Liste),
        wandleStundenInZahlen(Liste, ZahlenListe),
        flatten(ZahlenListe, ZahlenListeFlat),
        schreibeRelevanteVeranstaltungen(ZahlenListeFlat,
                        1000000, Ergebnis).
```

Da die Unterscheidung von Mehrfachstunden in Dreifach-, Vierfach- und Doppelstunden wichtig ist, kann bei der Auswertung nicht auf die schon indizierten Fakten aus der Vorverarbeitung zurückgegriffen werden, da in der Vorverarbeitung die Unterscheidung von Mehrfachstunden verloren geht. Vielmehr muss mit Hilfe von `wandleStundenInZahlen/2` die Umwandlung in Zahlen durchgeführt werden.

6.2.1.5 Veranstaltungen mit fest vorgegebenen Räumen

Veranstaltungen in einem bestimmten Raum oder Veranstaltungen, denen eine kleinere Raummenge vom Planersteller zugeordnet worden ist, sind schwieriger zu platzieren als Veranstaltungen mit freier Raumzuordnung.
Diese Veranstaltungen werden in einem nächsten Schritt herausgesucht und ihnen wird eine fiktive Teilnehmerzahl zugeordnet. Die Teilnehmerzahl aller Veranstaltungen geht mit in die Gewichtung der schwer zu platzierenden Veranstaltungen ein.

6.2.1.6 Restliche Veranstaltungen

Alle Veranstaltungen, die bei der Identifizierung der schwer platzierbaren Veranstaltungen noch nicht berücksichtigt worden sind, werden mit Hilfe des schon vorgestellten Prädikates `delayed_goals_number/2` in eine Reihenfolge gebracht und in dieser an das Ende der Platzierung angefügt.

6.2.2 Die Vermeidung ungünstiger Tage und Zeiten

Um zu vermeiden, dass eine Veranstaltung zu ungünstigen Zeiten platziert wird, werden diese Zeiten aus dem Wertebereich der Veranstaltungen entfernt. Der Programmcode zeigt die Umsetzung der Verkleinerung der Wertebereiche für den Sonnabend.

```
entferneSonnabend(Stundenplan, Index) :-
    findall(Tage, tag(Tage, _, _), TagListe),
    length(TagListe, M),
    % mehr Tage als eine 5-Tage-Woche?
    % -> die zusaetzlichen Tage nur bei Anforderung nutzen
    M > 5,
    AnzTage is 5,
    entferneSonnabend(Stundenplan, Index, AnzTage).

entferneSonnabend(_, 0, _).
entferneSonnabend(Stundenplan, Index, AnzTage) :-
    subscript(Stundenplan, [Index, 2], Tag),
    nonvar(Tag),
    IndexNeu is Index - 1,
    entferneSonnabend(Stundenplan, IndexNeu, AnzTage).
entferneSonnabend(Stundenplan, Index, AnzTage) :-
    subscript(Stundenplan, [Index, 2], Tag),
    % trotz der Zuweisung von 1..AnzTage an Tag,
    % werden nur die Tage zugewiesen, die sich
    % wirklich noch im Suchraum befinden.
    Tag :: 1..AnzTage,
    IndexNeu is Index - 1,
    entferneSonnabend(Stundenplan, IndexNeu, AnzTage).
```

Mit `nonvar(Tag)` wird durchgesetzt, dass die Reduzierung nur bei noch nicht auf einen `Tag` festgelegten Veranstaltungen erfolgt.

Bei der Reduzierung der Wertebereiche bleibt eine Einschränkung aus der Vorverarbeitung erhalten.

Ist der Sonnabend bzw. ein sechster Vorlesungstag in der Wissensbasis nicht vorgesehen, finden keine Berechnungen statt.

Für die Abendstunden verläuft die Reduzierung des Wertebereiches analog.

6.2.3 Die ausgewogene Auslastung aller Ressourcen

Um eine ausgewogene Auslastung der Räume, Tage und Zeiten zu erreichen, werden Arrays zu Verwaltungszwecken benutzt. Die gleichmäßige Auslastung wird auf die Woche bezogen erreicht und bezieht sich nicht auf die einzelnen Tage.

Um immer auf aktuelle Werte zugreifen zu können, muss diese Verwaltungsarbeit während des `labeling`-Prozesses stattfinden. Dazu wird die unter 5.2 vorgestellte einfachste funktionierende Planungskomponente erweitert.

```
labeling(_, [], _, _, _, _).
labeling(Stundenplan, [Index|Rest], IndexListe, Raeume,
     Tage, Zeiten) :-
    % Verwaltung
    veranstaltung(Index, Name, _, _, _),
    subscript(Stundenplan, [Index, 1], Raum),
    subscript(Stundenplan, [Index, 2], Tag),
    subscript(Stundenplan, [Index, 3], Zeit),
    sucheGuenstigstePlatzierung(Raum, Tag, Zeit, Raeume,
                 Tage, Zeiten),
    % Buchfuehrung
    anzRaeume(AnzRaeume),
    buchfuehren(Stundenplan, Raeume, Index, IndexListe,
            AnzRaeume, 1),
    anzTage(AnzTage),
    buchfuehren(Stundenplan, Tage, Index, IndexListe,
            AnzTage, 2),
    anzZeiten(AnzZeiten),
    buchfuehren(Stundenplan, Zeiten, Index, IndexListe,
            AnzZeiten, 3),
    labeling(Stundenplan, Rest, IndexListe, Raeume,
         Tage, Zeiten).
```

Die Intelligenz dieses Prädikates ist ausgelagert in `sucheGuenstigstePlatzierung/6`. Hier findet die Zuweisung mit Werten aus den Wertebereichen für `Raum`, `Tag` und `Zeit` mit Hilfe von `mein_indomain/2` statt.

```
sucheGuenstigstePlatzierung(Raum, Tag, Zeit, Raeume,
                 Tage, Zeiten) :-
    dom(Raum, RaumListe),
    erstelleSortierteListe(RaumListe, Raeume,
                SortierteRaumListe),
    mein_indomain(Raum, SortierteRaumListe),
    dom(Tag, TagListe),
    erstelleSortierteListe(TagListe, Tage,
                SortierteTagListe),
    mein_indomain(Tag, SortierteTagListe),
    dom(Zeit, ZeitListe),
    erstelleSortierteListe(ZeitListe, Zeiten,
                SortierteZeitListe),
    mein_indomain(Zeit, SortierteZeitListe).
```

`mein_indomain/2` arbeitet ähnlich wie die von `ECLiPSe` aufgestellte Regel `indomain` und initialisiert `Raum`, `Tag` und `Zeit` mit einem Wert der Domain. Dabei wird immer zuerst der Wert getestet, der im Suchablauf bisher am seltensten benutzt worden ist. Zur Auswahl dieses Wertes werden diese Arrays benötigt.

```
mein_indomain(DomainVar, _) :-
    nonvar(DomainVar).
mein_indomain(DomainVar, KriteriumsListe) :-
    var(DomainVar),
    dom(DomainVar, WertebereichsListe),
    sortiereWerte(WertebereichsListe, KriteriumsListe,
             SortierteWertebereichsListe),
    member(DomainVar, SortierteWertebereichsListe).
```

Arrays, die in `ECLiPSe` generiert werden, reagieren nicht logisch. Deshalb werden Arrays, die im Falle des `labeling`s verändert werden, beim Backtracking nicht berücksichtigt. Somit würde auf falsche Werte im `labeling`-Prozess zurückgegriffen werden.

Zwar bietet `ECLiPSe` die Möglichkeit, über das `built-in`-Prädikat `local reference/1` logische Arrays zu erzeugen. Sie sollten jedoch nicht benutzt werden, da die Zahl der Referenzen im System begrenzt ist. Außerdem werden dadurch die Lesbarkeit und die Wartbarkeit des Programms erschwert.

Deshalb werden die benötigten Arrays bei jedem Durchlauf von `labeling` neu initialisiert und mit Werten belegt. Dabei muss darauf geachtet werden, dass nur die Veranstaltungen berücksichtigt werden, die im `labeling`-Prozess schon bearbeitet worden sind.

Bei der Wertebelegung von `Raum`, `Tag` und `Zeit` der Veranstaltungen werden die Werte der Arrays genutzt, um die Sortierung der Wertebereiche zu beeinflussen.

So wird die gleichmäßige Auslastung von `Raum`, `Tag` und `Zeit` während der Woche erreicht.

6.3 Die Lösung

Bei einer korrekten Wissensbasis und einer geeigneten Wertebelegung für die Heuristik kommt die Planungskomponente schnell, effizient und mit wenigen Backtrackingschritten zu einer umsetzbaren und für die `Raum`-, `Tag`- und `Zeit`belegung optimierten Lösung.

Im erstellten Plan werden alle harten Einschränkungen berücksichtigt[2]. Dieser Plan ist an der Fachhochschule Wedel/PTL einsetzbar. Getestet wurde hauptsächlich auf einem PIII-400 mit 196MB RAM mit Linux. Der gesamte Durchlauf des Programmes benötigt auf diesem System knapp 10 Sekunden CPU-Zeit[3].

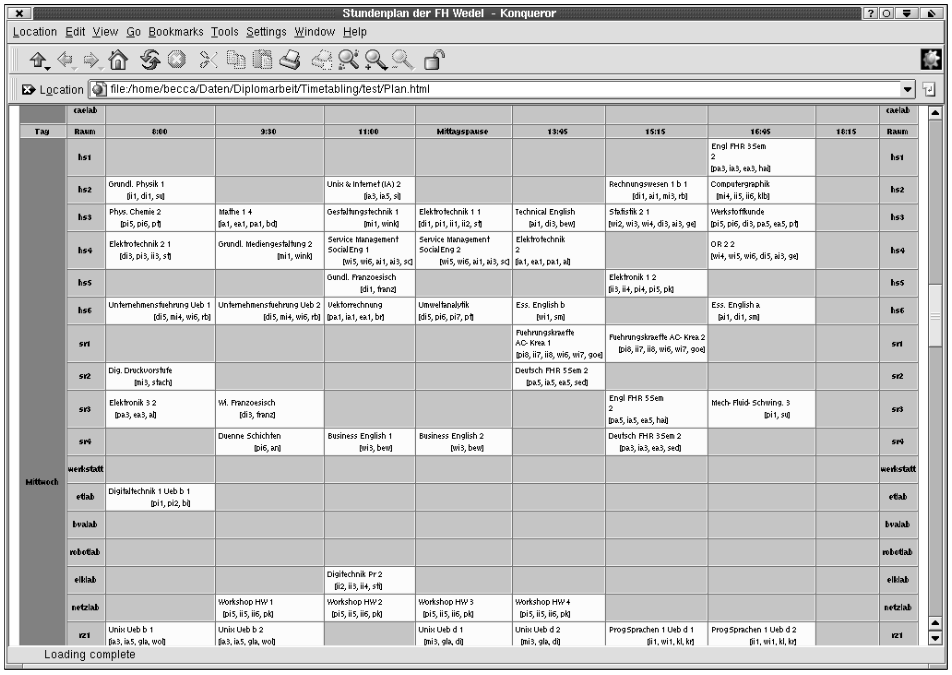

Abbildung 6.2: Ein Ausschnitt aus dem fertigen Plan unter Verwendung der Wissensbasis vom Wintersemester 1998/99

[2]Die im Plan eingegebenen Wünsche werden während der Planerstellung nicht berücksichtigt. Da die erstellte Planungskomponente ohne die Berechnung einer Qualitätsfunktion auskommt, sondern als Optimierungskriterium in diesem Fall die gleichmäßige Auslastung von `Raum`, `Tag` und `Zeit` genutzt wird, ist eine Durchsetzung der Wünsche qualitativ nicht messbar. Deshalb wird darauf verzichtet, die Wünsche zu bearbeiten.

[3]Bei CPU-Zeit handelt es sich um eine Zeitangabe, die von `ECLiPSe` bei einem erfolgreichen Durchlauf eines Programms ausgegeben wird.

7 Ergebnis und Ausblick

7.1 Ergebnis

Auf der Grundlage der Daten vom Wintersemester 1998/99 erweist sich das vorgestellte System als erfolgreich. Unter Verwendung der konstruktiven Suche wird ein gültiger Plan erstellt.

Das Ergebnis der Vorverarbeitung ist so zuverlässig, dass man sich bei der Entwicklung einer Planungskomponenten nur noch mit der Platzierung befassen muss. Alles andere - die Auswertung der Wissensbasis und die Berücksichtigung der Einschränkungen - übernimmt die Vorverarbeitung. Dadurch wird es leichter, sich auf das Wesentliche, die Platzierung, zu konzentrieren.

Das entstandene Programm ist wartbar, erweiterbar und modular. Neue Anforderungen können mit geringem Aufwand im System berücksichtigt werden. Änderungen müssen nur in der Vorverarbeitung vorgenommen werden und stehen dann allen weiteren Abläufen zur Verfügung.

7.2 Ausblick

Ein Programm dieses Umfangs ist immer ausbaufähig. Die Entwicklung kann selten als wirklich abgeschlossen bezeichnet werden. Aus diesem Grund sollen hier einige sinnvolle Erweiterungsansätze vorgestellt werden.

7.2.1 Bedienerfreundliche Benutzeroberfläche

Um von der breiten Masse der Stundenplanersteller akzeptiert zu werden, muss das System mit einer intuitiv zu bedienenden graphischen

Oberfläche ausgestattet werden, die dem Nutzer seine Arbeit erleichtert. Das Editieren von Dateien und das eigenständige Laden von Daten in `ECLiPSe` sollte Benutzern ohne Programmiererfahrung abgenommen werden.

Die meisten Fehler geschehen erfahrungsgemäß beim Schreiben der Wissensbasis. Es sind meistens nicht Fehler in der Logik, sondern einfache Tippfehler, die jedoch erhebliche Konsequenzen haben können.

Die von `ECLiPSe` ausgegeben Fehlermeldungen sind für den Laien nicht immer aussagekräftig. Es wäre wünschenswert, wenn dem Nutzer die Interpretation der Fehlermeldung durch die Benutzeroberfläche abgenommen würde.

Das Erstellen einer Oberfläche für das Stundenplanprogramm erfordert von einem Entwickler sehr viel Geschick, deshalb wäre es empfehlenswert, sich an vorhandenen Oberflächen von Stundenplansoftware zu orientieren.

Angenehm wäre es, die Dateneingabe sowohl über die Maus als auch über die Tatstatur zu erlauben. Bei der Umsetzung der Eingabe mit der Maus sollte es sich nicht um eine einfache 1-zu-1-Umsetzung der Prologfakten handeln. Vielmehr sollte dem Nutzer ein Plangerüst präsentiert werden, mit dem schon definierte Veranstaltungen per Maus fixiert werden können. So festgelegte Veranstaltungen sollten dann vom System in Prolog-Klauseln übertragen werden. Gleiches wäre für die Verfügbarkeit von Teilnehmern denkbar.

7.2.2 Ermöglichung manueller Eingriffe

Ein Stundenplanprogramm ist erst dann voll einsatzfähig, wenn es dem Ersteller möglich ist, dem Programm in letzter Instanz seinen Willen aufzuzwingen. Muss für solche Änderungen doch wieder die Stecktafel bemüht werden, sinkt die Akzeptanz des Programms drastisch.

Um Veranstaltungen innerhalb des Stundenplans zu verschieben, werden die vom System erstellten Einschränkungen benötigt, damit eine Veranstaltung nicht dort platziert wird, wo sie eigentlich nicht stattfinden kann. Solche Einschränken werden jedoch von `ECLiPSe` endgültig gelöscht, sobald einem Arrayelement ein Platz im Plan zugewiesen worden ist.

Ein einfaches Kopieren der Arbeitsstruktur löst das Problem nicht, da Einschränkungen, die ausschließlich im Speicher existieren, nicht mitkopiert werden können[1].

Die Lösung könnte so sein, dass in einer ersten Arbeitsstruktur ein vorläufiger Plan erstellt wird, und in einer zweiten Struktur könnten dann die Veranstaltungen im Nachhinein verschoben werden.

Ein weiterer Vorschlag zur Lösung dieses Problems wurde von Matthias Braisch angeregt. Die Fachhochschule Wedel/PTL plant zur Zeit die Anschaffung eines kommerziellen Stundenplanprogrammes. Das System verfügt über keinen ausgereiften Planungsalgorithmus und genügt daher nicht den Anforderungen, besitzt aber eine sehr gute, bewährte Oberfläche. Außerdem bietet das System die Möglichkeit, Daten aus fremden Systemen über eine Schnittstelle zu übernehmen. Die Überlegung läuft dahin, diese Schnittstelle mit den Daten des hier entwickelten Stundenplanungssystems zu speisen und eventuell manuelle Änderungen am fertigen Plan im gekauften System durchzuführen.

[1] Dieses Problem wird im Newsbeitrag `<3BFE9D17.B24AC5DD@icparc.ic.ac.uk>` diskutiert.

Anhang

A Prädikate der Wissensbasis

A.1 Die Stammdaten

In den Stammdaten werden die für den jeweiligen Plan unveränderlichen Daten festgehalten.

A.1.1 `ausgabeDatei('Datei')`

Diese Ausgabedatei, eine Prologdatei mit der Endung `pl`, enthält später den fertigen Plan.

Sie wird nach der Berechnung des Plans mit Daten gefüllt. Weitere Ausgabeprogramme, z.B. die Ausgabe des Planes in `HTML`, lesen diese Datei ein und erstellen anhand dieser den Ausgabeplan.

Beispiel: `ausgabeDatei('Plan.pl').`

A.1.2 `tag('Name', Kosten)`

`'Name'` ist der eindeutige Name eines Tages, der später bei der Berechnung des Stundenplans verwendet wird.

Bei `Kosten` handelt es sich um einen Wert, der bei einer kostenbasierten Lösungsfindung ausgewertet wird, um eine günstige Platzierung innerhalb des Plans zu erreichen. Hier kann jede beliebige Zahl eingesetzt werden. Eine günstige Kostenverteilung ist durch Erfahrung zu ermitteln.

Beispiel: `tag('Montag', 2).`

A.1.3 `zeit('Uhrzeit', Kosten)`

`'Uhrzeit'` ist der eindeutige Name einer Zeit, der später bei der Berechnung des Stundenplans verwendet wird.

`Kosten` werden analog zu A.1.2 definiert.

Beispiel: `zeit('9:30', 1).`

A.1.4 `raum(Name, Groesse, [Ressourcen], Kosten)`

Bei `Name` handelt es sich um den Raumnamen in Kleinbuchstaben ohne Leerzeichen.

`Groesse` beschreibt die Sitzplatzzahl im Hörsaal.

Bei `Ressourcen` handelt es sich um weitere Merkmale, die den Raum beschreiben. Hier werden neben der Wiederholung des Raumnamens, Beamer und andere Geräte erwähnt. Bei Räumen, in denen reguläre Vorlesungen stattfinden sollen, muss sich auch noch der Hinweis `vorlesung` in den Ressourcen befinden.

Wichtig ist, dass alle Ressourceninhalte mit einem Kleinbuchstaben beginnen.

`Kosten` werden analog zu A.1.2 definiert.

Beispiel: `raum(hs1, 60, [hs1, vorlesung, beamer], 0).`

A.1.5 `teilnehmer(Name, Groesse)`

Die einzelnen Teilnehmergruppen werden dem System über den Teilnehmerfakt bekanntgemacht.

Bei `Name` handelt es sich um eine eindeutige Bezeichnung in Kleinbuchstaben und ohne Leerzeichen.

`Groesse` bezeichnet die Teilnehmerzahl der beschriebenen Teilnehmergruppe.

Beispiel: `teilnehmer(wi3, 30).`

A.1.6 `dozenten([KuerzelListe])`

Unter Verwendung von `dozenten([KuerzelListe])` werden die Dozenten benannt. Dabei wird zwischen Assistenten und Dozenten nicht unterschieden. `KuerzelListe` wird in diesem Fall durch eine durch

Kommata getrennte Auflistung der Mitarbeiterkürzel in Kleinbuchstaben und ohne Leerzeichen ersetzt.

Beispiel: `dozenten([ha, si, sc, pt, klk, ahr, al, an]).`

A.1.7 untergruppe(UnterGruppenName, OberGruppenName, TeilnehmerName)

`UnterGruppenName` beschreibt den Namen der Untergruppe in Kleinbuchstaben und ohne Leerzeichen, der auch bei der Benennung von Veranstaltungen in den Vorlesungsdaten unter Teilnehmer verwendet wird.

`OberGruppenName` fasst mehrere Untergruppennamen zu einer gemeinsamen Gruppe zusammen. Auch dieser Name beginnt mit einem Kleinbuchstaben und enthält keine Leerzeichen.

An dritter Stelle dieses Fakts steht der `TeilnehmerName`, der Teilnehmer, auf den sich diese Untergruppe bezieht.

Das Untergruppenkonzept berücksichtigt automatisch, dass die Veranstaltungen der Gruppen mit gleichem `ObergruppenName` parallel laufen dürfen.

Beispiel: `untergruppe(wi1ps1, ps1ueb, wi1).`

A.2 Die Vorlesungsdaten

In dieser Datei werden die variablen Daten, die sich jedes Semester verändern, festgehalten.

A.2.1 veranstaltung('Name', [Teilnehmer], [Ressourcen])

Der Fakt `veranstaltung` wird benötigt, um alle Veranstaltungen eines Planes zu verwalten.

`'Name'` ist ein eindeutiger Name einer Veranstaltung in Anführungszeichen.

`Teilnehmer` enthält eine Liste mit Namen aus `teilnehmer(Name, Groesse)`, `untergruppe(UnterGruppenName, OberGruppenName, TeilnehmerName)` und `dozenten([KuerzelListe])`.

Bei `Ressourcen` handelt es sich um Auszüge aus der Ressourcenliste von `raum(Name, Groesse, [Ressourcen], Kosten)`. Handelt es sich bei der Veranstaltung um eine reguläre Vorlesung und nicht um eine Übung oder Praktikum, muss unter den Ressourcen `vorlesung` zu finden sein. Soll sich eine Einschränkung von Räumen auf mehrere Räume erstrecken, steht `oder([RaumListe])` zur Verfügung. Soll eine Veranstaltung zur selben Zeit in zwei (oder mehr Räumen) stattfinden, wird dieses mit `und([RaumListe])` modelliert.

Obwohl es an der Fachhochschule Wedel/PTL vier unterschiedliche Veranstaltungsarten gibt, werden sie nur durch zwei Prädikate im System dargestellt. Das eine Prädikat ist `veranstaltung('Name', [Teilnehmer], [Ressourcen])`, das andere, `terminNachAbsprache ('Name', [TeilnehmerListe])`, wird unter A.2.2.10 beschrieben.

`veranstaltung/3` wird benutzt, um Vorlesungen, Praktika, Übungen und Seminare zu definieren.

Um trotzdem für eine Unterscheidung zu sorgen, wird für eine Vorlesung bei den Veranstaltungsressourcen `vorlesung` eingefügt. Ein Praktikum bzw. eine Übung wird durch die Angabe eines speziellen Raumes (z.B. `rz1`), der als Ressourcenangabe unter `raum/4` die Ressource `vorlesung` nicht enthält, klassifiziert. Die Definition eines Seminars erfolgt analog zur Vorlesung.

Beispiel :

```
veranstaltung('Analysis 1 1', [pi1, ii1, di1, ha], [vorle-
sung]),
veranstaltung('Analysis 1 2', [pi1, ii1, di1, ha], [vorle-
sung]),
veranstaltung('Analysis 1 Ueb 1', [pi1, ii1, di1, ha],
[vorlesung]),
veranstaltung('Analysis 1 Ueb 2', [pi1, ii1, di1, ha],
[vorlesung]),
veranstaltung('Fertigungstechnik 2', [di5, wd],
[oder([hs3, hs4]), vorlesung]) und
veranstaltung('Elektronik', [di3, sz],
[und([hs3, etlab])]).
```

A.2.2 Die harten Einschränkungen

A.2.2.1 `doppelstunde('Veranstaltung1', 'Veranstaltung2')`

Mit diesem Fakt werden die Doppelstunden dem System bekanntgemacht.

`Veranstaltung1` und `Veranstaltung2` greifen auf Veranstaltungsnamen zu, die unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` deklariert worden sind.

Beispiel : `doppelstunde('Analysis 1 a 1', 'Analysis 1 a Ueb 1').`

A.2.2.2 `dreifachstunde('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3')`

Mit diesem Fakt werden die Dreifachstunden dem System bekanntgemacht.

`Veranstaltung1`, `Veranstaltung2` und `Veranstaltung3` greifen auf Veranstaltungsnamen zu, die unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` deklariert worden sind.

Beispiel : `dreifachstunde('Werkstattpraktikum 1', 'Werkstattpraktikum 2', 'Werkstattpraktikum 3').`

A.2.2.3 `vierfachstunde('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', 'Veranstaltung4')`

Mit diesem Fakt werden die Vierfachstunden dem System bekanntgemacht.

`Veranstaltung1`, `Veranstaltung2` , `Veranstaltung3` und `Veranstaltung4` greifen auf Veranstaltungsnamen zu, die unter `veranstaltung ('Name', [Teilnehmer], [Ressourcen])` deklariert worden sind.

Beispiel : `vierfachstunde('Workshop HW 1', 'Workshop HW 2', 'Workshop HW 3', 'Workshop HW 4').`

A.2.2.4 `teilnehmerNichtAmUm(Name, [Tage], [Zeiten])`

Mit diesem Fakt lassen sich Einschränkungen bezüglich der Teilnehmer formulieren.

`Name` steht dabei für den Namen des Teilnehmers, der unter `teilnehmer(Name, Groesse)` oder `dozenten([KuerzelListe])` definiert worden ist.

`Tage` steht für eine Liste von Tagen unter `tag('Name', Kosten)` in einfachen Anführungsstrichen und durch Kommata getrennt. Eine fehlende Einschränkung wird durch einen Unterstrich(_) gekennzeichnet.

`Zeiten` steht für eine Liste von Zeiten unter `zeit('Name', Kosten)` in einfachen Anführungsstrichen und durch Kommata getrennt. Wie bei `Tage` bezeichnet auch hier der Unterstrich (_) eine fehlende Einschränkung.

Beispiel : `teilnehmerNichtAmUm(si, ['Dienstag'], ['13:45', '15:15', '16:45', '18:15']).`

A.2.2.5 `veranstaltungAmUmIn('Name', [Tage], [Zeiten], Raum)`

Mit diesem Fakt wird festgelegt, an welchem Tag, zu welcher Zeit und in welchem Raum eine Veranstaltung stattzufinden hat.

`Name` steht für den Namen einer Veranstaltung, die unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert worden ist.

`[Tage]` und `[Zeiten]` werden wie unter A.2.2.4 behandelt.

`Raum` wird bei einer Auswertung der Wissensbasis nicht beachtet und wird deshalb hier nicht belegt. Der `Raum` an dieser Stelle wird beim Schreiben des Ergebnisses mit einem Wert belegt.

Beispiel : `veranstaltungAmUmIn('Phys. Grundl. Pr 1', ['Montag'], ['13:45'], _).`

A.2.2.6 `veranstaltungNichtAmUmIn('Name', [Tage], [Zeiten], Raum)`

Mit diesem Fakt wird festgelegt, an welchem Tag, zu welcher Zeit und in welchem Raum eine Veranstaltung nicht stattfinden kann.

`Name` steht für den Namen einer Veranstaltung, die unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert wurde.

`[Tage]` und `[Zeiten]` werden wie unter A.2.2.4 berücksichtigt.

Die `Raum`behandlung wird wie unter A.2.2.5 beschrieben durchgeführt.

Beispiel : `veranstaltungNichtAmUmIn('Marketing 1', ['Montag', 'Dienstag', 'Mittwoch', 'Samstag'], _, _).`

A.2.2.7 `veranstaltungenAmSelbenTag('Veranstaltung1', 'Veranstaltung2'), veranstaltungenAmSelbenTag('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3'), veranstaltungenAmSelbenTag('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', 'Veranstaltung4')`

Dieser Fakt sorgt dafür, dass zwei Veranstaltungen auf jeden Fall am selben Tag stattfinden.

`Veranstaltung1`, `Veranstaltung2`, `Veranstaltung3` und `Veranstaltung4` stehen für Namen, die vorher unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert worden sind.

Beispiel : `veranstaltungenAmSelbenTag('Gundl. Franzoesisch', 'Wi. Franzoesisch').`

A.2.2.8 `veranstaltungenNichtAmSelbenTag('Veranstaltung1', 'Veranstaltung2'), veranstaltungenNichtAmSelbenTag('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3'), veranstaltungenNichtAmSelbenTag('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', 'Veranstaltung4')`

Dieser Fakt sorgt dafür, dass zwei Veranstaltungen auf keinen Fall am selben Tag stattfinden.

`Veranstaltung1`, `Veranstaltung2`, `Veranstaltung3` und `Veranstaltung4` stehen für Namen, die vorher unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert worden sind.

Beispiel : `veranstaltungenNichtAmSelbenTag('BWL 1', 'BWL 2').`

A.2.2.9 `veranstaltungenNichtZusammenhaengend('Veranstaltung1', 'Veranstaltung2')`

Dieser Fakt sorgt dafür, dass zwei Veranstaltungen nicht nacheinander stattfinden können.

`Veranstaltung1` und `Veranstaltung2` stehen für Namen, die vorher unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert worden sind.

Beispiel : `veranstaltungenNichtZusammenhaengend('BWL 1', 'BWL 2').`

A.2.2.10 `terminNachAbsprache('Name', [TeilnehmerListe])`

Mit Hilfe dieses Fakts werden Veranstaltungen definiert, die während der Planung nicht berücksichtigt werden. Erst für die Ausgabe des Plans in PROLOG bzw. in einer anderen Ausgabesprache werden diese Veranstaltungen ausgewertet.

Beispiel : `terminNachAbsprache('Informatik-Seminar', [wi5, ii5, mi5, si]).`

A.2.3 Die Wünsche

Die Wünsche beschreiben Einschränkungen, die berücksichtigt werden können, aber nicht berücksichtigt werden müssen, um einen funktionierenden Plan zu erstellen. Die Berücksichtigung eines Wunsches steigert die Qualität eines Planes. Wünsche, die mit harten Einschränkungen konkurrieren, werden in keinem Fall berücksichtigt. Die Priorität wird durch eine positive Zahl ausgedrückt. Je höher diese Zahl ist, desto wichtiger ist dem Teilnehmer die Berücksichtigung des Wunsches.

A.2.3.1 `doppelstunde('Veranstaltung', 'Veranstaltung2', Prioritaet)`

Mit diesem Fakt werden gewünschte Doppelstunden einbezogen.

`Veranstaltung1` und `Veranstaltung2` greifen auf Veranstaltungsnamen zu, die unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` deklariert worden sind.

Beispiel : `doppelstunde('Analysis 1 a 1', 'Analysis 1 a Ueb 1', 10).`

A.2.3.2 `dreifachstunde('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', Prioritaet)`

Mit diesem Fakt werden gewünschte Dreifachstunden eingegeben.
`Veranstaltung1`, `Veranstaltung2` und `Veranstaltung3` greifen auf Veranstaltungsnamen zu, die unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert worden sind.

Beispiel : `dreifachstunde('Werkstattpraktikum 1', 'Werkstattpraktikum 2', 'Werkstattpraktikum 3', 6).`

A.2.3.3 `vierfachstunde('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', 'Veranstaltung4', Prioritaet)`

Mit diesem Fakt werden gewünschte Vierfachstunden einbezogen.
`Veranstaltung1`, `Veranstaltung2` , `Veranstaltung3` und `Veranstaltung4` greifen auf Veranstaltungsnamen zu, die unter `veranstaltung ('Name', [Teilnehmer], [Ressourcen])` deklariert worden sind.

Beispiel : `vierfachstunde('Workshop HW 1','Workshop HW 2', 'Workshop HW 3', 'Workshop HW 4', 20).`

A.2.3.4 `teilnehmerNichtAmUm(Name, [Tage], [Zeiten], Prioritaet)`

Mit diesem Fakt lassen sich Wünsche zur Verfügbarkeit der Teilnehmer formulieren.
`Name` steht dabei für den Namen des Teilnehmers, der unter `teilnehmer(Name, Groesse)` oder `dozenten([KuerzelListe])` definiert wurde.
`[Tage]` und `[Zeiten]` werden analog zu A.2.2.4 definiert.

Beispiel : `teilnehmerNichtAmUm(si, ['Dienstag'], _, 15).`

A.2.3.5 `veranstaltungAmUmIn(Name, [Tage], [Zeiten], Raum, Prioritaet)`

Mit diesem Fakt wird festgelegt, an welchem Tag und zu welcher Zeit eine Veranstaltung stattzufinden hat.
`Name` steht für den Namen einer Veranstaltung, die unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert wurde.

`[Tage]` und `[Zeiten]` sind analog zu A.2.2.4 definiert.
Für die Raumbelegung gilt die Anmerkung zu A.2.2.5.

Beispiel : `veranstaltungAmUmIn('Phys. Grundl. Pr 1', ['Montag'], ['13:45'], _, 12).`

A.2.3.6 `veranstaltungNichtAmUmIn(Name, [Tage], [Zeiten], Raum, Prioritaet)`

Mit diesem Fakt wird festgelegt, an welchem Tag, zu welcher Zeit und in welchem Raum eine Veranstaltung nicht stattzufinden hat.

`Name` steht für den Namen einer Veranstaltung, die unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert wurde.

`[Tage]` und `[Zeiten]` sind analog zu A.2.2.4 definiert.
Für die Raumbelegung gilt die Anmerkung zu A.2.2.5.

Beispiel : `veranstaltungNichtAmUmIn('Marketing 1', ['Montag', 'Dienstag', 'Mittwoch', 'Samstag'], _, _, 8).`

A.2.3.7 `veranstaltungenAmSelbenTag('Veranstaltung1', 'Veranstaltung2', Prioritaet), veranstaltungenAmSelbenTag('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', Prioritaet), veranstaltungenAmSelbenTag('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', 'Veranstaltung4', Prioritaet)`

Dieser Fakt sorgt dafür, dass zwei Veranstaltungen am selben Tag stattfinden.

`Veranstaltung1`, `Veranstaltung2`, `Veranstaltung3` und `Veranstaltung4` stehen für Namen, die vorher unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert worden sind.

Beispiel : `veranstaltungenAmSelbenTag('Gundl. Franzoesisch', 'Wi. Franzoesisch', 30).`

A.2.3.8 `veranstaltungenNichtAmSelbenTag('Veranstaltung1', 'Veranstaltung2', Prioritaet)`, `veranstaltungenNichtAmSelbenTag('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', Prioritaet)`, `veranstaltungenNichtAmSelbenTag('Veranstaltung1', 'Veranstaltung2', 'Veranstaltung3', 'Veranstaltung4', Prioritaet)`

Dieser Fakt sorgt dafür, dass zwei Veranstaltungen nicht am selben Tag stattfinden.

`Veranstaltung1`, `Veranstaltung2`, `Veranstaltung3` und `Veranstaltung4` stehen für Namen, die vorher unter `veranstaltung('Name', [Teilnehmer], [Ressourcen])` definiert worden sind.

Beispiel: `veranstaltungenNichtAmSelbenTag('BWL 1', 'BWL 2', 5).`

B Benutzerhandbuch

B.1 Programminstallation

Um das Programm zur Stundenplanerstellung nutzen zu können, wird die Expertensystementwicklungsumgebung `ECLiPSe` benötigt. Diese ist gemäß der Installationshinweise zu installieren.

B.1.1 Linux

Das Stundenplanerstellungsprogramm wird in ein beliebiges Verzeichnis (z.B. `$HOME/Stundenplansystem`) kopiert. Zum Starten von `ECLiPSe` wechselt man in dieses Verzeichnis und ruft `tkeclipse &` auf.

B.1.2 Windows

Hier wird das Stundenplanerstellungsprogramm in das Arbeitsverzeichnis von ECLiPSe kopiert. Dieses liegt unter dem Installationsverzeichnis in `lib_tcl`. Zum Starten von `ECLiPSe` wird danach auf das von `Windows` angelegte Programmicon geklickt.

B.2 Programmstart

Das Stundenplanerstellungssystem wird mit dem Aufruf `start (Stammdaten, Vorlesungsdaten)` gestartet. Dabei sind `Stammdaten` und `Vorlesungsdaten` durch die entsprechenden Dateinamen zu ersetzen. Ein Beispielaufruf kann `start('Stammdaten.pl', 'Vorlesungsdaten.pl')` lauten, mit den Dateien `Stammdaten.pl` und `Vorlesungsdaten.pl`.

Der Stundenplan wird nun nach den Vorgaben berechnet.

B.3 Aufbau der Stamm- und Vorlesungsdaten

Die Eingabe der Daten geschieht als Prologfakten. Jeder Fakt steht in einer neuen Zeile, jede Zeile wird mit einem Punkt(.) beendet. Einzeilige Kommentare werden mit „%“ gekennzeichnet, mehrzeilige werden mit „/*“ geöffnet und mit „*/“ geschlossen.
Weitere Hinweise zur Erstellung der Datenbasis sind Anhang A zu entnehmen.

B.4 Fehlerbehandlungen

Das Programm ist beim Durchlauf in der Lage, eindeutige Fehler zu erkennen. Greifen zum Beispiel als `doppelstunde` gekennzeichnete Veranstaltungen nicht auf als `veranstaltung` gekennzeichnete Veranstaltungen zu, bricht das Progamm ab.
Bei erkennbaren Fehlern handelt es sich nur um Tippfehler und einfachere Probleme. Auf keinen Fall ist das Programm in der Lage, komplexere Fehleingaben zu erkennen und zu melden.

B.5 Das Testmodul

- **Testen, ob die Veranstaltungen in den zur Verfügung stehenden Zeiteinheiten platziert werden können ...**

  Tritt hier ein Fehler auf, wurde versucht, zu viele Veranstaltungen in dem Zeitraum

  $$AnzahlTage * AnzahlZeiten * AnzahlRäume \tag{B.1}$$

  zu platzieren.
  Mögliche Fehlerursachen: Es wurden unter `tag`, `raum`, `zeit` zu wenige Tage, Räume und Zeiten deklariert.

- **Testen, ob alle Veranstaltungsnamen eindeutig sind ...**

  Bei einem Fehler an dieser Stelle wurde für zwei Veranstaltungen derselbe Name vergeben. Veranstaltungsnamen müssen jedoch eindeutig sein.
  Fehlerbehebung: Umbennenen der genannten Veranstaltungen, so dass keine Konflikte mehr auftreten.

- **Testen, ob alle Mehrfachstunden auf reale Veranstaltungen zugreifen ...**

  Bei Mehrfachstunden handelt es sich um `doppelstunde`, `dreifachstunde` und `vierfachstunde`. Die dabei benutzten Veranstaltungsnamen müssen bei `veranstaltung` genannt werden.
  Fehlerbehebung: Umbenennen der genannten Veranstaltungen in real existierende.

- **Testen, ob Fehler in der Raumzuteilung sind ...**

  An dieser Stelle wird getestet, ob einer Veranstaltung mehr als ein fester Raum zugeordnet wird, ohne `oder` zu benutzen.
  Fehlerbehebung: Zusammenfassen der Räume der Veranstaltung mit Hilfe von `oder`.

- **Testen, ob einer Veranstaltung Räume anhand der gewünschten Ressourcen zugeordnet werden können ...**

  Hier wird versucht, den Veranstaltungen anhand der Ressourcen einen bestimmten Raum zuzuweisen. Ist dies nicht möglich, wird ein Fehler gemeldet.
  Fehlerbehebung: Überprüfen der zugeteilten Ressourcen der genannten Veranstaltungen.

- **Testen, ob „Karteileichen“ geführt werden ...**

  Wird hier eine Teilnehmergruppe (Dozenten oder Studenten) ausgegeben, dann werden diese zwar als Teilnehmer geführt, nehmen aber an keiner Veranstaltung teil.
  Fehlerbehebung: Entfernen der Teilnehmergruppen aus der Wissensbasis. Diese Teilnehmer verbrauchen unnötige Rechenzeit.

- **Testen, ob alle Vorlesungen eines Teilnehmers in den zur Verfügung stehenden Zeiteinheiten platziert werden können...**

  Manche Teilnehmer haben Tage und Zeiten, zu denen sie nicht an Veranstaltungen teilnehmen möchten. Diese Zeiten werden in `teilnehmerNichtAmUm` erfaßt. Hier wird nun überprüft, ob der Teilnehmer mit diesem Einschränkungen noch in der Lage ist, seinen Teilnahmeverpflichtungen nachzukommen.
  Fehlerbehebung: Einschränkungen des Teilnehmers lockern.

- **Testen, ob alle weiteren Veranstaltungen auf reale Veranstaltungen zugreifen ...**

  An dieser Stelle werden alle weiteren, noch nicht getesteten Veranstaltungsnamen (`veranstaltungAmUmIn`, `veranstaltungNicht-AmUmIn`, `veranstaltungenNichtZusammenhaengend`, `veranstal-tungenAmSelbenTag`, `veranstatlungenNichtAmSelbenTag`) überprüft.
  Fehlerbehebung: Umbennenen der genannten Veranstaltungen in real existierende.

Weitere mögliche Fehler

Sollte das Programm nicht in der Lage sein, eine Lösung zu finden, kann dies daran liegen, dass die Werte für die Konfiguration der Planungskomponente falsch gewählt worden sind. In diesem Fall wird kein Fehler gemeldet und das Programm nicht abgebrochen, da `ECLiPSe` nicht in der Lage ist, einen Fehler zu erkennen.

Es kann auch sein, dass Fehler in der Wissensbasis vom Testmodul nicht erkannt worden sind. Liegt der Fehler im Bereich der Vorverarbeitung, gibt `ECLiPSe` eine nicht sehr aussagekräftige Fehlermeldung aus und bricht das Programm ab.

C Daten für das Knapsackproblem

Um das unter 2.5 erwähnte Beispiel ablaufen lassen zu können, werden einige Testdaten benötigt. Die Datensätze `datenFuerRucksackProblem/5` unterscheiden sich nur im ersten Parameter. Es wird hochgezählt von 1 bis 100. Über diese Datensätze wird die Zahl der Durchläufe für die lokale Suche gesteuert.

```
:- mode datenFuerRucksackProblem(+,-,-,-,-).

datenFuerRucksackProblem(1, 20,
           [2, 97, 62, 100, 16, 71, 26, 92, 15, 68, 90, 35, 110, 69, 18,
47, 12, 54, 47, 43],
           [1, 96, 67, 90, 13, 74, 22, 86, 23, 63, 89, 25, 100, 76, 23,
56, 5, 47, 45, 39], 100).
datenFuerRucksackProblem(2, 20,
           [2, 97, 62, 100, 16, 71, 26, 92, 15, 68, 90, 35, 110, 69, 18,
47, 12, 54, 47, 43],
           [1, 96, 67, 90, 13, 74, 22, 86, 23, 63, 89, 25, 100, 76, 23,
56, 5, 47, 45, 39], 100).

...

datenFuerRucksackProblem(99, 20,
           [2, 97, 62, 100, 16, 71, 26, 92, 15, 68, 90, 35, 110, 69, 18,
47, 12, 54, 47, 43],
           [1, 96, 67, 90, 13, 74, 22, 86, 23, 63, 89, 25, 100, 76, 23,
56, 5, 47, 45, 39], 100).
datenFuerRucksackProblem(100, 20,
           [2, 97, 62, 100, 16, 71, 26, 92, 15, 68, 90, 35, 110, 69, 18,
47, 12, 54, 47, 43],
           [1, 96, 67, 90, 13, 74, 22, 86, 23, 63, 89, 25, 100, 76, 23,
56, 5, 47, 45, 39], 100).
```

Literaturverzeichnis

Allgemein

[kop96] H. Kopka. LaTeX Einführung Band 1 (2. überarbeitete Auflage). Addison-Wesley (Deutschland) GmbH, Bonn 1996.

[goo00] M. Goossens, F. Mittelbach, A. Samarin. Der LaTeX Begleiter. Addison-Wesley (Deutschland) GmbH, Bonn 2000.

[dud93] Bibliographisches Institut Mannheim & F.A. Brockhaus AG. DUDEN - Informatik, 2.vollständige und erweiterte Auflage. Dudenverlag, Mannheim 1993.

[knu97] D.E. Knuth. The Art of Computer Programming, Sorting and Searching, Second Edition. Addison-Wesley Longman, 2000.

[htm98] S. Münz. SELFHTML, Version 7.0, 1998. http://www.teamone.de/selfaktuell/index.html

[due90] G. Dueck, T.Scheuer. Journal of Computer Physics, Nr. 90, 1990, S.161.

[rei01] S. Reith, Dr. H. Vollmer. Wer wird Millionär? Komplexitätstheorie: Konzepte und Herausforderungen. c't 7/01, S. 240.

Timetabling und Scheduling

[coo00] T.B. Cooper, J.H. Kingston. The Complexity of Timetable Construction Problems, Basser Department of Computer Science. The University of Sidney, Australia 2000.

[dip99] A. Rieck. Automatische Stundenplangenerierung mittels Constraint Logic Programming und der Expertensystementwicklungsumgebung `ECLiPSe`. Fachhochschule Wedel, 1999/2000.

[sem01] M. Braisch. Stundenplangenerierung mit CLP. Fachhochschule Wedel, 2001.

[ass00] M. Nitz, A. Ehlers, M. Lütgens. Test und Beurteilung des Stundenplansystems von André Rieck. Assistenz an der Fachhochschule Wedel, 2000.

[sta98] P. Stamatopoulus, E. Vigalis, S. Karaboyas. Nearly Optimum Timetable Construction through CLP and intelligent Search. University of Athens, Department of Informatics, 1998.

[bur97] E.K. Burke, L.P. Newall, R.F. Weare. A Simple Heuristically Guided Search for the Timetable Problem. University of Nottingham, Automated Scheduling and Planning Group, 1997.

[abd98] S. Abdennadher, M. Marte. Constraintbasierte Stundenplanung von Universitäten. Ludwig-Maximilians-Universität, München, Institut für Informatik, 1998.

[rud99] H. Rudová, L. Matyska. Timetabling with Annotations. Masaryk University, Faculty of Informatics, 1999.

[aze94] F. Azevedo, P. Barahona. Timetabling in Constraint Logic Programming. Universidade Nova de Lisboa, Departemento de Informática, 1994.

[sch95] A. Schaerf. A survey of automated timetabling. Stichting Mathematisch Centrum, Amsterdam 1995. CS-R9567.

Expertensysteme und Künstliche Intelligenz

[ric91] E. Rich, K. Knight. Artificial Intelligence, Second Edition. McGraw-Hill, 1991.

[rus95] S. Russel, P. Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, New Jersey 1995.

Prolog

[clo87] W.F. Clocksin, C.S. Mellish. Programming in Prolog, Third edition. Springer Verlag, New York 1987.

[oke90] R.A. O'Keefe. The Craft of Prolog. The MIT Press, England 1990.

[ste97] L. Sterling, E. Shapiro. The Art of Prolog, Second Edition. The MIT Press, England 1997.

[bra01] I. Bratko. Prolog - Programming for Artificial Intelligence, Third Edition. Pearson Education Limited, Essex 2001.

[wei97] W. Weisweber. Prolog - Logische Programmierung in der Praxis. International Thomson Publishing, Bonn 1997.

[die91] F.-R. Diepenbrock. Prolog-Werkzeugkasten. Hanser, München, Wien 1991.

[bla01] P. Blackburn, J.Bos, K. Striegnitz. Learn Prolog Now!. 2001
http://www.coli.uni-sb.de/∼kris/learn-prolog-now

Constraints

[mar98] K. Marriott, Peter J. Stuckey. Programming with Constraints: An Introduction. Massachusetts Institute of Technology, USA 1997.

[fru93] T. Frühwirth et al. Constraint Logic Programming - An Informal Introduction. Technical Report ECRC-93-5.

[wal95] M. Wallace. Constraint Programming. IC-Parc, 1995.

[all92] D. McAllester. Constraint Satisfaction Search. Lecture Notes for 6.824, Artificial Intelligence, 1992.

[wal97] M. Wallace et al. ECLiPSe: A Platform for Constraint Logic Programming. IC-Parc, 1997.

[bar98] R. Bartak. On-line Guide to Constraint Programming. Universita Karlova v Praze, 1998. http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html

[pou95] D. Pountin. Constraint Logic Programming. Februarausgabe des BYTE Magazins 1995.

Dokumentation zu ECLiPSe

[tut99] J. Schimpf, K. Shen, M. Wallace. Tutorial on Search in `ECLiPSe`. Imperial College, London 1999.

[lib00] W. Harvey et al. Library Manual. International Computers Limited and ECRC GmbH, 1990-1995 und International Computers Limited and Imperial College London, 1996-2000.

Relevante Newsgroups

- comp.constraints
- comp.lang.prolog
- comp.ai
- sci.op-research
- sci.logic

Danksagung

Ich bedanke mich bei Prof. Dr. Uwe Schmidt für die vorbildliche Betreuung.

Matthias Braisch danke ich für die konstruktive Zusammenarbeit, Arne Albrecht und Egon Kühlke für die liebevolle Versorgung während einer arbeitsreichen Zeit und Thorsten Ehlers für die sehr gute Einführung in `xml` und `xsl`.

Das Korrekturlesen wurde liebevoll von meinem Freund Thorsten, seinem Bruder Arne, meinem Onkel Günther und meinen Eltern übernommen.

Danke.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Hamburg, 25. Januar 2002 Unterschrift