INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians-
Universität
München

Ein constraint-basierter Dienstplaner für Krankenstationen

Hans Schlenker

Diplomarbeit

Beginn der Arbeit:	6. Februar 1997
Abgabe der Arbeit:	5. August 1997
Aufgabensteller:	Prof. Dr. François Bry
Betreuer:	Dipl.-Inform. Slim Abdennadher

Hiermit versichere ich, daß ich diese Arbeit selbstständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Eine Kurzform dieser Arbeit habe ich zusammen mit Slim Abdennadher veröffentlicht [AS97]. Daraus sind Teile des Textes in den vorliegenden Bericht übernommen, die Stellen sind nicht gekennzeichnet.

München, am 5. August 1997 Hans Schlenker

Danksagung

Ich möchte an dieser Stelle Slim Abdennadher, François Bry, Klaus Dässler, Ellen Lilge, Peter Richter, Norbert Trapp und vor allem Karin ganz herzlich danken. Ohne jeden einzelnen von ihnen wäre diese Arbeit so nicht möglich gewesen. Außerdem danke ich der Siemens Nixdorf Informationssysteme AG für ihre freundliche Unterstützung.

München, am 5. August 1997 Hans Schlenker

Inhaltsverzeichnis

Kapitel 1

Einleitung

Constraints sind Relationen, die es erlauben, in deklarativer Weise Probleme durch Angabe von Nebenbedingungen zu formulieren, die im Vergleich zu Logikprogrammiersprachen die Suche nach geeigneten Lösungen beschleunigen [Van89, FM94]. Die Constraints werden dann nicht mittels Resolution, sondern durch spezielle Algorithmen gelöst. Die Programme, durch die diese Algorithmen implementiert werden, nennt man Constraint-Löser.

Mit der Einbettung von Constraints in Logikprogrammiersprachen wurde es möglich, schnell und elegant komplexe kombinatorische Probleme durch eine Verbindung aus Constraint-Erfüllung und Suche zu lösen. Konkret ergeben sich vor allem folgende Vorteile:

- deklarative Modellierung unter Verwendung problemnaher Constraints
- automatische Propagierung impliziter Effekte während der Suche nach einer Lösung
- Trennung der Constraint-Behandlung vom eigentlichen Suchverfahren

Seit Anfang der neunziger Jahre wird constraint-basierte Programmierung mit großem Erfolg von mehreren Firmen weltweit kommerziell eingesetzt. Anwendungsgebiete sind Produktions- und Ressourcenplanung (insbesondere Zeit- und Kapazitätswirtschaft), Transportoptimierung, Layoutgenerierung, CAD-Systeme und Personalplanung. Dazu gehört die Generierung von Dienstplänen für Krankenstationen, die heute noch Gegenstand der Forschung ist.

Immer noch werden an den meisten Krankenhäusern die Dienstpläne von Hand erstellt. Dabei wird für jede Schwester oder jeden Krankenpfleger[1] einer Station und für jeden Tag des Planungszeitraums – üblicherweise ein Monat – eine oder keine Schicht zugeteilt. Eine Schicht ist ein zeitlicher Abschnitt eines Tages. Ein Tag ist in Schichten eingeteilt, damit die

[1] im folgenden nennen wir ein Mitglied des Pflegepersonals immer *Schwester*.

Station rund um die Uhr mit Personal besetzt werden kann. Für die Zuteilung der Schichten müssen verschiedene „harte“ Bedingungen unbedingt erfüllt werden, und „weiche“ Bedingungen nach Möglichkeit. Eine solche Planungstätigkeit ist, speziell bei größeren Krankenstationen, sehr aufwendig: typischerweise benötigt eine Person einen ganzen Tag zur Planung eines Monats.

Das hier dargestellte System bildet Teile der händischen Dienstplan-Erzeugung nach. Damit kann ein mit der Dienstplanung vertrauter Benutzer, in einer Krankenstation die sogenannte *Stationsschwester*, interaktiv innerhalb einiger Minuten einen Plan erstellen. Wir erwarten, dass dieses System

- vom Benutzer eher akzeptiert wird, als ein vollautomatisches System, da er an der unmittelbaren Planung beteiligt bleibt,
- leicht und schnell zu einem korrekten Plan führt, da der Benutzer nicht sein komplettes Wissen über die Planung dem System mitteilen muss, und dadurch auch
- bessere Lösungen als ein vollautomatisches System und oft sogar als die händische Generierung liefert, und damit
- akzeptabel ist für das von der Planung betroffene Personal.

Das Verfahren wurde im Rahmen dieser Arbeit prototypisch implementiert und heisst INTERDIP.

Vor der eigentlichen Planung muss der Benutzer die konkreten Rahmenbedingungen für den Monat spezifizieren. Das geschieht mit grafischen Eingabewerkzeugen. Ein Dienstplan wird dann mit INTERDIP in mehreren Phasen generiert. In jeder Phase werden nach vorgegebenen Mustern jeweils mehrere Tage des Dienstplans auf einmal belegt. Dadurch wird inkrementell ein Dienstplan erzeugt, der nach Beendigung aller Phasen vollständig ist.

Durch die Generierung in Phasen und Mustern erreichen wir im Vergleich mit existierenden Verfahren eine enorme Reduzierung des Suchraums, wodurch gute Ergebnisse für realistische Krankenstationen (20 Schwestern) innerhalb einiger Minuten erzeugt werden können.

INTERDIP ist ein Prototyp, mit dem tatsächlich Dienstpläne erzeugt werden können. Damit hat der Benutzer ein Werkzeug, mit dem er zum einen unter Anwendung der gegebenen Möglichkeiten fast alle Rahmenbedingungen der Planung berücksichtigen kann und zum anderen durch geschickte Ausnutzung der interaktiven Möglichkeiten gute Pläne bekommt. Insgesamt verfolgen wir damit also einen pragmatischen Ansatz, der nicht sämtliche Funktionen, die Dienstplanung betreffen, vollständig automatisch erfüllen will. Unter anderem ist keine Planung eines ganzen Jahres, mit Urlaubsplanung und Stundenverrechung implementiert. Vielmehr wurde

Wert darauf gelegt, zu zeigen, wie mit den aktuell verfügbaren Constraint-Techniken ein Verfahren implementiert werden kann, das dem Dienstplaner einer Krankenstation hilft, schneller und leichter zu einem korrekten Plan zu kommen.

Das Verfahren und die Implementierung ist möglichst allgemein gehalten. So sind die Phasen homogen in dem Sinne, dass alle gleich funktionieren und sich nur in Daten unterscheiden. Das lässt eine einfache Erweiterung um zusätzliche Phasen für andere Schichtmodelle (konkrete Einteilung eines Tags in Schichten) zu. Die Darstellung der Muster ist flexibel und mit ihnen kann der Benutzer starken Einfluss auf die konkrete Belegung das Planes in den einzelnen Phasen nehmen. Die globalen Einschränkungen, ein Teil der Rahmenbedingungen, sind ausdruckstark: mit ihnen können relativ einfach komplexe Zusammenhänge formuliert werden. Auch dadurch läßt sich INTERDIP leicht erweitern.

Implementiert wurde INTERDIP mit IF/Prolog [Sie96b] der Siemens-Nixdorf-Informationssysteme AG, das basierend auf den Forschungsarbeiten von CHIP [DVS+88] ein Constraint-Paket enthält [Sie96a].

Die Arbeit entstand in Zusammenarbeit mit einer Stationsschwester eines Krankenhauses in München. Die dort gebräuchliche Dienstplanung wird in einschlägigen Kursen unterrichtet und ist damit auch an anderen Krankenhäusern üblich. Sie ist allerdings insofern speziell, als bei der Planung die Wünsche der Schwestern starke Berücksichtigung finden. Das ist nicht überall üblich. Die Rahmenbedingungen stammen aus einer konkreten Station, sind aber ansonsten typisch. INTERDIP ist im übrigen so allgemein formuliert, dass eine Anpassung an andere Gegebenheiten problemlos möglich ist.

Dieses Papier ist wie folgt gegliedert: Kapitel 2 beschreibt das Problem der Dienstplangenerierung und Kapitel 3 dessen Modellierung als *Constraint Satisfaction Problem*. In Kapitel 4 sehen wir, wie Dienstpläne von Hand erzeugt werden und unseren daraus abgeleiteten inkrementellen Ansatz. In Kapitel 5 ist die Implementierung von INTERDIP beschrieben. Anschließend folgt das Handbuch: Kapitel 6. Kapitel 7 schließlich enthält Zusammenfassung und Ausblick. Im Anhang ist der komplette Programmtext von INTERDIP gegeben.

Für die Benutzung dieser Dokumentation wird vom Leser die Kenntnis der Programmiersprache Prolog erwartet [SS94].

Eine Kurzfassung dieser Darstellung wird im Rahmen des 12. Workshops logische Programmierung – WLP'97 veröffentlicht [AS97].

Kapitel 2

Beschreibung des Problems

Dieses Kapitel beschreibt das Problem der Dienstplanung für Krankenstationen und den technischen Rahmen, in dem die vorliegende Arbeit stattfand.

2.1 Dienstpläne für Krankenstationen

Die Erzeugung von optimalen Dienstplänen ist ein komplexes Problem. Für Krankenstationen sieht es wie folgt aus.

In regelmäßigen zeitlichen Abständen, meist jeden Monat, muss für jede Station eines Krankenhauses ein neuer Dienstplan erstellt werden. Dabei müssen gesetzliche Bestimmungen, dienstliche Regelungen, Personaldaten und nicht zuletzt Wünsche des Personals berücksichtigt werden. Alle diese Parameter geben den Rahmen für einen guten Dienstplan.

Eine Station eines Krankenhauses ist eine organisatorische Einheit. Eine bestimmte Station hat gewisse spezialisierte Aufgaben zu übernehmen, verfügt über Räume im Krankenhaus und über Personal. Die Stationen eines Krankenhauses sind meist völlig voneinander getrennt[1], jede hat ihre eigenen Räume und ihr eigenes Personal. Deshalb behandeln wir hier Krankenstationen getrennt voneinander, das heißt, wir betrachten genau *eine* Krankenstation.

Der Dienstplan eines Monats ist eine Zuordnung des Personals der Krankenstation zu den Schichten der Tage dieses Monats. Eine Schicht ist ein Zeitabschnitt: jeder Tag hat die Einheiten *Frühschicht*, *Spätschicht* und *Nachtschicht* und unter Umständen weitere. Jeder Schicht jedes Tages muss Personal zugeordnet werden.

Abbildung 2.1 zeigt einen Dienstplan für zehn Schwestern und einen Zeitraum von 14 Tagen. Jede Zeile der Tabelle steht für eine Schwester, jede

[1] Jedenfalls gilt das zum Zeitpunkt der Entstehung dieser Arbeit. Das hat seine Gründe nicht zuletzt in der heutigen Unmöglichkeit, Dienstpläne für ein ganzes Krankenhaus zu erstellen. Es ist also durchaus möglich, dass in Zukunft Krankenstationen organisatorisch stärker verbunden werden.

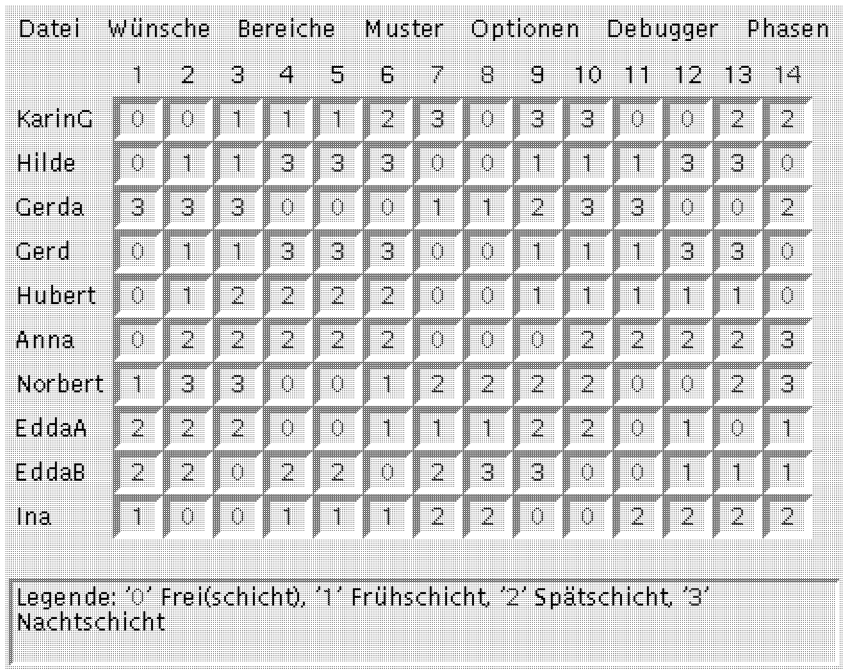


Abbildung 2.1: Beispiel eines Dienstplans für 10 Schwestern und 14 Tage

Spalte für einen Tag. Jede Zelle der Tabelle ist also genau einer Schwester und genau einem Tag zugeordnet. In ihr steht, ob und wann die Schwester an diesem Tag arbeitet. Dabei steht *0* für *Freischicht*, *1* für *Frühschicht*, *2* für *Spätschicht* und *3* für *Nachtschicht*.

Für die Zuordnung des Personals zu den Arbeitsschichten gibt es verschiedene Rahmenbedingungen. Grob kann man diese in vier Arten unterteilen: gesetzliche Bestimmungen, dienstliche Regelungen, Personaldaten und Wünsche des Personals.

2.1.1 Gesetze

Die wichtigsten Anforderungen, die an Dienstpläne gestellt sind, kommen von Gesetzen und Tarifverträgen.

So ist tarifvertraglich geregelt, dass im Pflegedienst seit dem 1.4.1990 im Schnitt wöchentlich 38,5 Stunden gearbeitet wird ([Sch95, §15]). Tatsächlich muss die wöchentliche Arbeitszeit im Schnitt eines halben Jahres 38,5 Stunden betragen. Anders gesagt: wird die regelmäßige wöchentliche Arbeitszeit in einer Woche über- oder unterschritten, dann kann das innerhalb 26 Wochen ausgeglichen werden. Zusammen mit anderen Regelungen ergibt die wöchentliche eine regelmäßige monatliche Arbeitszeit, die möglichst einzuhalten ist.

Der Tarifvertrag regelt auch die Ruhezeit: §15 schreibt eine Mindestruhezeit von 11 Stunden vor[2]. Dadurch kann erstens einer Schwester maximal eine Schicht pro Tag zugeordnet werden. Zweitens gilt für ein typisches Modell mit drei Schichten (siehe nächster Abschnitt): arbeitet eine Schwester an einem Tag in der Nachtschicht, so darf sie am nächsten Tag weder Früh- noch Spätschicht leisten, das heißt, sie muss am nächsten entweder wieder nachts arbeiten, oder aber einen freien Tag haben. Für dasselbe Schichtenmodell gilt: auf eine Spätschicht darf keine Frühschicht folgen.

Mutterschutzurlaub ist gesetzlich geregelt (Mutterschutzgesetz, siehe auch [Sch95]). Unter anderem dürfen *werdende oder stillende* Mütter nicht Nachts und an Sonn- und Feiertagen beschäftigt werden. Außerdem steht ihnen nach der Geburt eine freie Zeit (Mutterschutzurlaub) zu, während der die Station ohne sie auskommen muss.

Tarifvertraglich geregelt ist auch, dass nach Aufstellung eines Urlaubsplans, und das wird heute häufig gemacht, eine *einseitige Änderung des Urlaubstermins grundsätzlich nicht mehr möglich ist* ([Sch95, S. 313]). Insofern ist die Urlaubsplanung von der Dienstplanung unabhängig. Urlaubstage stehen (im wesentlichen) schon vorher fest und können im Prinzip nicht bei der Dienstplanung bestimmt werden.

Werden Arbeitnehmer an einem Sonntag beschäftigt, müssen sie einen Ersatzruhetag haben, der innerhalb eines [...] Zeitraums von zwei Wochen zu gewähren ist (Arbeitszeitgesetz, siehe [Sch95, S. 157]). Diese Bedingung kann vereinfacht durch die Einschränkung „maximal 10 Arbeitstage hintereinander“ ausgedrückt werden (siehe Abschnitt 6.28). Dasselbe Gesetz sagt: es *müssen mindestens 15 Sonntage im Jahr beschäftigungsfrei bleiben.*

2.1.2 Dienstliche Regelungen

Dienstliche Regelungen sind solche, die für ein spezielles Krankenhaus, einen Teil des Krankenhauses oder auch nur für eine Station gelten. Sie werden von der jeweiligen Leitung festgesetzt und sind meist organisatorischer Art.

Dienstlich geregelt wird das Schichtenmodell der Station. Üblicherweise definieren in einem Krankenhauses Personalrat und Pflegedienstdirektion mehrere Schichtmodelle, die sich in der Anzahl und Länge der Schichten unterscheiden. Die Schichtmodelle haben natürlich die einschlägigen Gesetze zu berücksichtigen. Jede Station sucht sich daraus dann ein zu den individuellen Gegebenheiten der Station passendes aus. Ein solches, das wir im folgenden *das typische* Modell nennen wollen, ist folgendes Drei-Schichten-Modell: *Frühschicht* von 6.15 bis 14.45 Uhr, *Spätschicht* von 13.30 bis 22.00 Uhr und *Nachtschicht* von 21.30 bis 6.45. Jede Schicht schließt 30 Minuten Pause ein. Damit dauern Früh- und Spätschicht 8 und die Nachtschicht 8,75

[2]Von der 11-Sunden-Regel sind zwar auch Ausnahmen möglich, diese spielen aber in der Praxis meist keine Rolle

Stunden. Eine beliebte Erweiterung dieses Modells ist um eine *Zwischenschicht* von 9.00 bis 17.30 Uhr zum Ausgleich eventueller Engpässe in Früh- und Spätschicht.

Auch die Grösse einer Station (Anzahl der Schwestern) ist eine dienstliche Regelung. Die Leitung eines Krankenhauses legt zusammen mit den Krankenkassen die Größe jeder Station fest. Zusammen mit den tariflichen Arbeitszeiten, dem Schichtenmodell und Erfahrungswerten zum durchschnittlichen Krankheitsausfall ergibt die Größe der Station Mindest- und Höchstbelegungen für jede Schicht jedes Tags. Wir benutzen im folgenden eine typische Station mit 20 Schwestern und Mindestbelgungen von: 3 Schwestern in der Frühschicht, 3 in der Spätschicht und 2 in der Nachtschicht.

Eine dienstliche Regelung ist auch die Dienstplanung selbst. Dazu gehört, inwieweit Wünsche der Schwestern berücksichtigt werden, und, damit zusammenhängend, ob in regelmäßigen Schichtfolgen gearbeitet wird, oder jeder Monat individuell geplant. Wir nehmen hier an, dass Wünsche starke Berücksichtigung finden und deshalb individuell geplant wird (das ist die konkrete Situation in dem Krankenhaus, mit dem im Rahmen dieser Arbeit zusammengearbeitet wurde).

2.1.3 Personaldaten

Personaldaten geben die individuellen Rahmenbedingungen für eine Person. Dazu zählen vor allem die vertraglich feststehende monatliche Arbeitszeit (z.B. Teilzeit), noch ausstehender Urlaub und angefallene Überstunden. Alle diese Werte fließen ja in die konkrete Planung eines Monats ein.

2.1.4 Wünsche

Wünsche sind schließlich Bedingungen die das Personal selbst formuliert.

Typische Wünsche sind:

Freiwünsche Der häufigste Wunsch: eine Schwester möchte an einem bestimmten Tag frei haben.

Positive/Negative Schichtwünsche Eine Schwester möchte an einem Tag in einer bestimmten Schicht arbeiten bzw. nicht arbeiten.

Sympathie / Antipathie Eine Schwester möchte bzw. möchte nicht zusammen mit einer bestimmten anderen Schwester in einer Schicht arbeiten.

Kombinierte Wünsche „Wenn ich am n-ten Tag des Monats arbeiten muss, dann möchte ich am m-ten Tag frei haben“.

Vernünftige Schichtfolgen Gerne wird in der üblichen Folge „fünf Tage arbeiten, zwei Tage frei" gearbeitet. Beliebt sind auch homogene Schichtfolgen: es ist angenehmer, an aufeinanderfolgenden Tagen in der gleichen Schicht zu arbeiten, als jeden Tag in einer anderen Schicht.

2.2 Korrekte und gute Dienstpläne

Dienstpläne müssen korrekt sein und sollten gut sein.

Korrekt ist ein Dienstplan, wenn er alle Bedingungen erfüllt, die er erfüllen muss. Einige der Bedingungen, die oben genannt wurden, gehören dazu, vor allem die Belegungsgrenzen der Station in den einzelnen Schichten, die Arbeitszeiten der Schwestern oder vereinbarter Urlaub. Diese Bedingungen nennen wir auch *harte Bedingungen*.

Wir haben oben aber auch Bedingungen gesehen, die nach Möglichkeit erfüllt werden sollen: vor allem Wünsche der Schwestern. Das sind die *weichen Bedingungen*.

Leider kann nicht jeder Dienstplan immer alle Bedingungen erfüllen. Insbesondere die Wünsche der Schwestern führen oft zu Widersprüchen. Man denke nur an den verbreiteten Wunsch, am Wochenende und an Feiertagen frei zu haben. Zusammen mit einer Mindestbelegung der Station führt das schnell zu einem Widerspruch.

Ein guter Dienstplan ist ein korrekter Dienstplan, der möglichst viele möglichst wichtige weiche Bedingungen erfüllt. Dazu müssen die weichen Bedingungen gewichtet sein: siehe auch die Abschnitte 6.16.1 und 6.45.1.

2.3 Technischer Rahmen dieser Arbeit

Diese Arbeit entstand zum Teil bei der Siemens Nixdorf Informationssysteme AG (SNI) und wurde von dieser freundlich unterstützt. Die Zusammenarbeit fand unter einigen technischen Voraussetzungen statt, die den Inhalt der Arbeit beeinflussten.

SNI hatte geplant, aufbauend auf ihr Produkt IF/Prolog, einem Prolog-Entwicklungssystem (siehe Abschnitt 5.1), eine handvoll Problemlöser zu entwickeln und zu vermarkten. Dabei sollte ein „generischer" Ansatz verfolgt werden. Das heißt, es sollte ein Baukasten entwickelt werden, mit dem konkrete Problemlöser, vor allem im Bereich Ressourcenplanung, realisiert werden können. Das sollten kleine, handliche Programme sein, die für wenig Geld in grossen Stückzahlen verkauft werden sollten und auf möglichst weit verbreiteten Computersystemen zu laufen hätten.

INTERDIP ist eines der ersten konkreten Projekte dieser Planungen.

Ziel dieser Arbeit war es also, mit IF/Prolog, eventuell unter Erweiterung des Systems, einen Dienstplaner für Krankenhäuser zu entwickeln und prototypisch zu realisieren. Dabei war auf die Übertragbarkeit dabei entstehender Ansätze auf andere, ähnliche Probleme, Wert zu legen. Der endgültige Generator sollte darüberhinaus auf MS-Windows-Systemen in vernünftiger Zeit Dienstpläne erstellen können.

Kapitel 3

Modellierung

Dieses Kapitel beginnt mit einer allgemeinen Beschreibung von Constraint-Problemen (engl. Constraint Satisfaction Problem, CSP). Darauf folgt eine Definition des Begriffs der Komplexität solcher Probleme. Dann wird eine fast kanonische Modellierung des Dienstplanproblems als CSP gegeben und schließlich im Rahmen dieser Arbeit untersuchte andere Modellierungen beschrieben.

3.1 Constraint-Probleme

3.1.1 CSP (Constraint Satisfaction Problem)

Ein Constraint-Problem (engl. *Constraint Satisfaction Problem*, CSP) kann wie folgt [Van91, S. 3f] definiert werden.

Ein CSP besteht aus Variablen, Wertebereichen dafür, und Constraints. Gegeben sei eine endliche Menge I von Variablen $\{X_1, X_2, ..., X_n\}$, die ihre Werte respektive aus ihren Wertebereichen $D_1, D_2, ..., D_n$ annehmen, und eine Menge von Constraints. Ein Constraint $c(X_{i_1}, X_{i_2}, ..., X_{i_k})$ zwischen k Variablen ist eine Teilmenge des Kartesischen Produkts $D_{i_1} \times D_{i_2} \times ... \times D_{i_k}$, die angibt, welche Werte der Variablen zueinander passen. In der Praxis müssen diese Teilmengen nicht explizit angegeben werden, sondern können mittels Gleichungen, Ungleichungen oder irgendwelchen Prädikaten definiert werden. Eine Lösung eines CSP ist eine Zuweisung von Werten zu allen Variablen, so daß alle Constraints erfüllt sind.

Da Constraints eine formale oder technische Beschreibung von Bedingungen sind, werden beide Begriffe häufig simultan verwendet, insbesondere nennen wir im folgenden *Bedingungen* oft auch *Constraints*.

3.1.2 PCSP (Partial Constraint Satisfaction Problem)

Oft haben reale Probleme sehr viele Constraints, die insgesamt widersprüchlich sind. Das Problem heißt dann auch *over-constrained*. Ein solches

Problem, als CSP formuliert, hat keine Lösung. Dafür braucht es eine Verallgemeinerung von CSP: *Partial Constraint Satisfaction Problems*, PCSP. Ein PCSP ist ein CSP, nur muss eine Lösung eines PCSP nicht alle Constraints erfüllen. Die Constraints, die erfüllt werden müssen, heißen *harte Constraints* und die anderen *weiche Constraints* oder englisch *Hard Constraints* bzw. *Soft Constraints*.

Übrigens: obwohl PCSP eine Verallgemeinerung der CSP sind, ist doch üblich, im allgemeinen von CSP zu sprechen. *CSP* ist die gebräuchliche Kurzform von *Constraint-Problem*.

3.1.3 Lösen von Constraint-Problemen

Generate-and-Test

Das Finden einer Lösung für ein CSP ist ein Suchproblem. Für Suchprobleme ist die einfachste Methode *generate-and-test*.

Dabei wird zuerst eine vollständige Variablenbelegung, das ist eine Belegung aller Variablen, generiert. Dann wird getestet, ob diese Belegung alle Constraints erfüllt. Sind alle Constraints erfüllt, ist eine Lösung des CSP gefunden. Sind nicht alle Constraints erfüllt, wird im nächsten Schritt eine neue, andere Variablenbelegung erzeugt und diese getestet. Sind alle möglichen Variablenbelegungen erzeugt worden, ohne dass eine alle Constraints erfüllt hätte, ist das CSP unlösbar.

Insgesamt werden also, bis eine Lösung gefunden ist, alle möglichen vollständigen Variablenbelegungen generiert und getestet.

Constraint-Solving

Generate-and-Test ist zwar korrekt, aber extrem ineffizient. Deshalb wurden intelligentere Methoden zur Lösung von CSPs entwickelt, die allgemeinste, die auch Techniken älterer Ansätze umfasst, ist *Constraint-Solving*.

Die Idee des Constraint-Solving ist es, alle Constraints des CSP maximal aufzulösen und implizite Effekte, die sich aus der Art der Constraints einerseits und aus den Wertebereichen der Variablen andererseits ergeben, zu propagieren und dadurch neue Constraints zu erhalten. Dafür existieren effektive spezielle Algorithmen [Kum92, VSD93]. Die Programme, durch die diese Algorithmen implementiert werden, nennt man *Constraint-Löser* oder *Constraint-Solver*. Die Constraints werden im Zuge der Abarbeitung eines Programmes inkrementell dem sogenannten Constraint-Speicher hinzugefügt. Dem Constraint-Löser obliegen u.a. die Aufgaben, den Constraint-Speicher bei Hinzukommen neuer Constraints zu aktualisieren und zu vereinfachen und zu prüfen, ob die im Speicher vorliegenden Constraints schon ausreichen, den Wert einer Variable eindeutig zu bestimmen.

Im optimalen Fall genügen allein die Constraints zum Finden einer Lösung des CSP mit einem Constraint-Löser. Ist das nicht möglich, ist auch hier Suche notwendig. Auch die wird aber durch Constraint-Lösen optimiert: während der Suche einer Lösung, also durch Belegen oder auch durch Einschränken von Wertebereichen von Variablen, ändert sich das CSP und es können Effekte propagiert und neue Constraints erzeugt werden. Insgesamt wird dadurch der Suchraum während der Suche eingeschränkt.

Soft-Constraint-Solving

Für die Behandlung weicher Bedingungen gibt es in der Literatur verschiedene Ansätze, unter anderem Hierarchische Constraints [BMMW89], Possibilistische Constraints [Sch92], Constraint Relaxation [WHG96] und Branch-and-Bound [FW92]. Soft-Constraint-Solving verwendet fast immer eigene Constraint-Solver. Einzig eine Abart der Hierarchischen Constraints und die Branch-and-Bound-Technik ist mit Standard-Constraint-Solving (nur harte Constraints) vereinbar.

Für den Ansatz der Hierarchischen Constraints ist eine Hierarchie der Constraints nötig. Dazu müssen alle Constraints des Problems in Klassen eingeteilt werden, die eine Hierarchie darstellen: von harten über wichtige zu weniger wichtigen Constraints. Dann wird zuerst eine Lösung gesucht, die alle Constraints der „harten Klasse“ erfüllt. Gelingt das, wird eine Lösung gesucht, die alle der „harten Klasse“ und der nächstwichtigen Klasse erfüllt. Und so weiter. Problematisch an diesem Verfahren ist die Einteilung der Constraints in Klassen unterschiedlicher Wichtigkeit. Das Verfahren impliziert, dass jeder Constraint einer Klasse wichtiger ist, als alle der nächsten, weniger wichtigen Klasse, zusammen. Und eine solche Einteilung ist praktisch selten zu finden. Insbesondere für die Dienstplanung ist das fast unmöglich.

Flexibler ist da Branch-and-Bound, zusammen mit einem Constraint-Löser, der ausschließlich harte Constraints behandeln kann. Die harten Constraints beschreiben die korrekten Lösungen. Der Constraint-Solver behandelt also nur die harten Constraints und sorgt für eine korrekte Lösung. Korrekte Lösungen werden dann anhand einer Kostenfunktion, in die die Verletzung der weichen Constraints einfließt, optimiert. Diese Technik verwenden wir bei INTERDIP.

3.1.4 Constraint-Solving und Prolog

CSP bestehen aus Variablen, Wertebereichen und Constraints. Die Variablen eines CSP werden in Prolog direkt durch Prolog-Variablen repräsentiert[1]. Wertebereiche und Constraints werden mit Prädikaten definiert. Die Definition eines Constraints in Prolog ist der Aufruf eines ent-

[1]Wo im folgenden eine sprachliche Unterscheidung nötig ist zwischen den Variablen, die das CSP definieren, und anderen Prolog-Variablen, nennen wir erste auch *Domain-Variablen*.

sprechenden Prädikats. Dieses fügt den Constraint im Constraint-Speicher ein, was wiederum den in Prolog eingebauten Constraint-Solver zu einer Reaktion (Vereinfachung, Propagation) veranlasst. Der Constraint-Solver weiß auch, für welche Variablen Wertebereiche und Constraints definiert sind und kann auf deren Veränderung durch das Programm reagieren. Sobald das CSP inkonsistent ist (durch neue Constraints oder während der Suche nach einer Lösung durch eine Variablenbelegung), scheitert das zugehörige Prädikat.

Den Aufruf eines Constraint-Prädikats nennen wir im folgenden auch *Vereinbarung*, *Erklärung*, *Formulierung* oder *Definition* des Constraints.

3.1.5 FD-Constraints

Constraint Solving passiert in der Praxis innerhalb sogenannter *Constraint-Systeme* [FA97]:

Die Art der Constraints und der Wertebereiche bestimmen das Constraintsystem. Beispiele für Constraint-Systeme sind lineare Gleichungssysteme über reellen Zahlen, die Boolesche Algebra, Rechnen mit endlichen Wertebereichen oder die Manipulation von Zeichenketten.

Das Constraint-System zum Rechnen mit endlichen Wertebereichen heißt auch *Finite-Domain-Constraints* oder *FD-Constraints*. Für diese Arbeit wichtig sind nur die FD-Constraints. Obwohl wir am Ende nur Variablen mit booleschen Wertebereichen benutzen, ist das Boolesche Constraintsystem für unsere Anwendung weniger geeignet. Wir benötigen zur Formulierung der Constraints nämlich die Summe über Variablenwerte, und die kann im Booleschen Constraintsystem nicht ausgedrückt werden.

Im folgenden sind die wichtigsten FD-Constraints als IF/Prolog-Prädikate dargestellt. Dabei ist ein *Term* eine (Prolog-)Variable, eine ganze Zahl, ein Ausdruck der Form `+`*Term*, ein Ausdruck der Form `-`*Term*, ein Ausdruck der Form *Term* `+` *Term*, ein Ausdruck der Form *Term* `-` *Term*, ein Ausdruck der Form *Term* `*` *Term*, oder ein Ausdruck der Form `(`*Term*`)`. Ein *Term* ist also eine Art arithmetischer Term und der Wert eines Terms ist entsprechend.

`A ?= B` $\longleftarrow$[2] der Wert des Terms `A` ist gleich dem Wert des Terms `B`.

`A ?< B` $\longleftarrow$ der Wert des Terms `A` ist kleiner dem Wert des Terms `B`.

`A ?=< B` $\longleftarrow$ der Wert des Terms `A` ist kleiner oder gleich dem Wert des Terms `B`.

`A ?> B` $\longleftarrow$ der Wert des Terms `A` ist größer dem Wert des Terms `B`.

`A ?>= B` $\longleftarrow$ der Wert des Terms `A` ist größer oder gleich dem Wert des Terms `B`.

[2] Hier benutze ich eine beliebte Abkürzung für „trifft genau dann zu, wenn“: $\longleftarrow$

`A ?\= B` ⟵ der Wert des Terms `A` ist ungleich dem Wert des Terms `B`.

Das sind die einfachen sog. *arithmetischen* Constraints. Daneben gibt es sogenannte *globale* Constraints (Constraints bezüglich mehrerer Variablen):

`all_distinct(VarList)` ⟵ die Werte aller Variablen aus `VarList` sind verschieden.

`cardinality(Min,Max,CondList)` ⟵ `CondList` ist eine Liste *arithmetischer* Constraints und mindestens `Min` und höchstens `Max` davon sind erfüllt. Ist `Min` oder `Max` das Atom '`*`', so gibt es keine Unter- bzw. Obergrenze.

`domain_if(Condition,ThenGoal,ElseGoal)` ⟵ `Condition` ist ein arithmetischer Constraint und sobald entschieden ist, ob dieser Constraint erfüllt ist oder nicht, wird `ThenGoal` bzw. `ElseGoal` ausgeführt. (Konditional)

`domain_if(Condition,ThenGoal)` ⟵ `domain_if(Condition,ThenGoal,true)`. (Implikation)

Hier ist zu beachten, dass die Vereinbarung eines Constraints für bestimmte Variablen, also der Aufruf eines Constraint-Prädikats, in der Regel nicht direkt die entsprechende Bedingung überprüft, in der Regel werden Constraints ja für ungebundene Variablen formuliert. Vielmehr veranlasst die Vereinbarung den Constraint-Solver, diese Bedingung für die beteiligten Variablen einzutragen.

Ein Constraint-System benötigt auch eine Möglichkeit, Wertebereiche zu definieren. In IF/Prolog geschieht das mit dem Prädikat `in/2`[3]:

`V in L` ⟵ `V` ist eine Variable oder eine Liste von Variablen und für diese ist der Wertebereich `L` definiert. `L` ist dabei eine geordnete Liste von ganzen Zahlen.

3.2 Komplexität

Die Probleme, die üblicherweise als CSP oder PCSP formuliert werden, sind sehr komplex. Es handelt sich um kombinatorische Probleme: eine Lösung davon ist eine Kombination von Variablenbelegungen, die bestimmte Bedingungen erfüllen muss und andere erfüllen soll. Diese Probleme gehören meist zu der Klasse der NP-vollständigen Probleme, deren prominentester Vertreter das Erfüllbarkeitsproblem aussagenlogischer Formeln ist (siehe [CS93]).

[3] Eine verbreitete Art, Prolog-Prädikate eindeutig zu identifizieren, ist die Nennung des Namens und der Stelligkeit: `Name/Stelligkeit`.

Wir nennen die Menge aller möglichen Variablenbelegungen den *Lösungsraum* L eines CSP. Natürlich (das heißt: in aller Regel) ist nicht jedes Element des Lösungsraums auch eine Lösung des CSP, doch ist jede Lösung des CSP ein Element des Lösungsraums. Mit obigen (siehe 3.1.1) Bezeichnungen gilt:

$$L = D_1 \times D_2 \times ... \times D_n$$

Die Grösse[4] $|L|$ des Lösungsraums ist:

$$|L| = |D_1| * |D_2| * ... * |D_n|$$

Wir nennen die Menge aller Möglichkeiten, die ein Suchverfahren zur Lösung eines CSP im schlechtesten Fall versuchen müsste, den *Suchraum* S. Wir nennen die Größe $|S|$ des Suchraums auch *Komplexität* C.

Zur Suche nach einer Lösung würde ein naives Suchverfahren (generate-and-test), wenn auch nur im schlechtesten Fall, alle möglichen Belegungen aller Variablen erzeugen und prüfen und damit wäre der Suchraum gleich dem Lösungsraum. Aber auch für die Verfahren mit Constraint Solving, bei denen ja der Suchraum während der Suche nach einer Lösung „eingeschnitten“ wird, existiert keine allgemeine Theorie über den Unterschied zwischen Such- und Lösungsraum. Nur für Spezialfälle sind da bis heute Aussagen getroffen worden. Wir sprechen im folgenden von der „Komplexität eines Problems“, wenn wir kein bestimmtes Suchverfahren meinen. Dabei wird immer $L = S$ als obere Schranke des Suchraums angenommen. Wird ein konkretes Suchverfahren angegeben, bezieht sich die Komplexität auf den konkreten Suchraum.

Der Ansatz bei der Modellierung von INTERDIP war, den Lösungsraum einzuschränken, um damit den Suchaum von vorneherein zu begrenzen. Zusätzlich geben wir aber auch ein Suchverfahren an, das den Suchraum gegenüber dem Lösungsraum weiter einschränkt.

3.3 Modellierung des Dienstplanproblems als CSP

Das Dienstplanproblem kann folgendermassen als CSP formuliert werden.

Wie wir in Kapitel 2 gesehen haben, ist ein Dienstplan eine Tabelle mit einer Zeile für jede Schwester und einer Spalte für jeden Tag. In jeder Zelle des Plans steht entweder eine '0', was bedeutet, die entsprechende Schwester hat an dem Tag frei, oder aber eine Schicht, die die Schwester an dem Tag zu leisten hat.

Vor dem ersten Ansatz müssen wir das Problem der Freischichten klären.

[4] Wenn M eine Menge ist, dann ist $|M|$ die Kardinalität der Menge.

3.3.1 Freischichten

Wenn in einer Zelle des Dienstplans eine Null steht, bedeutet das, die Schwester hat an dem Tag frei. Der Schwester mag genügen, nur zu wissen, *dass* sie frei hat, und nicht, *warum* sie frei hat. Für die Dienstplanung ist es aber ein großer Unterschied, ob eine Schwester Urlaub hat, ob sie Mutterschutz genießt und deshalb Mutterschaftsurlaub hat, ob sie einen Tag Sonntagsausgleich bekommt, oder aber einfach frei hat, weil sie trotz dieses freien Tages ihr Stundensoll im Monat erfüllt.

Ein Dienstplanungsprogramm muss entsprechend den freien Tag der Schwester als Arbeitstag gutschreiben (Urlaub), dafür sorgen, dass die Schwester auch tatsächlich frei bekommt (z.B. Mutterschutz), einer Schwester nach Möglichkeit Sonntagsausgleich zukommen lassen und diesen unter Umständen auf alle Schwestern gleichmäßig verteilen oder aber einen freien Tag als Null geleistete Stunden verrechnen.

Dementsprechend müsste ein solches Planugsprogramm für jede mögliche Art von Freischichten einen anderen Eintrag im Dienstplan vornehmen, um den dann korrekt verrechnen zu können. Schon in [WH95] wird, basierend auf der Arbeit [Fre91], nachgewiesen, dass die Werte für die Freischichten auf einen reduziert werden können. Die Autoren benutzten eine Modellierung P des Dienstplanproblems als CSP und verwenden dafür n Variable, die alle denselben Wertebereich D haben. Zwei Werte $i, j \in D$ heißen *austauschbar*, falls jede Lösung des CSP, die i enthält, eine Lösung bleibt, falls i durch j ersetzt wird und umgekehrt jede Lösung die j enthält eine Lösung bleibt, falls j durch i ersetzt wird. P ist dann in ein Problem P' so überführbar, dass die Wertemenge D durch die Wertemenge $D' = D \setminus \{j\}$ ersetzt wird und eine Lösung von P' unmittelbar in eine Lösung von P übersetzt werden kann und P genau dann lösbar ist, wenn P' lösbar ist. Dadurch reduziert sich die Komplexität von $C_P = |D|^n$ auf $C_{P'} = (|D| - 1)^n$.

Die Autoren zeigen, dass in ihrer Modellierung des Dienstplanproblems alle Arten von Freischichten durch einen Wert repräsentiert werden kann. Damit reduziert sich die Komplexität konkret von 10^n auf 4^n (siehe nächster Abschnitt).

Genaugenommen aber nehmen die Autoren in ihrer Arbeit vereinfachend an, dass z.B. Urlaub genau wie jede andere Freischicht behandelt werden kann. Insbesondere verrechnen sie Urlaub als Null geleistete Stunden. Wir werden die Freischichten aber auf verschiedene Arten behandeln. Trotzdem können wir die Ergebnisse von [WH95] benutzen.

3.3.2 Variablen und Werte

Für die Modellierung als CSP benötigen wir zunächst die Darstellung der Variablen und deren Wertebereiche. Die geläufigste Modellierung (siehe u.a. [WH95, HW96, LA96]) ist die folgende.

Wir verwenden für jede Zelle des Dienstplans eine Variable. Diese Variable bekommt dann als Wert den Eintrag in den Dienstplan zugewiesen, hat also als Wertebereich $\{0, 1, 2, 3\}$ für Freischicht, Frühschicht, Spätschicht und Nachtschicht, respektive. Für die typische Größe (siehe Abschnitt 2.1.2) eines Dienstplans hat dieses Problem aber eine enorme Komplexität:

$$C = 4^{20*30} \approx 10^{361}$$

Im folgenden benutzen wir für die Bezeichnung der Variablen eine Prolog-ähnliche Notation mit Metavariablen. Wir schreiben für die Anzahl der Schwestern s, für die der Tage t und für eine Variable des CSP `V`ij, wobei i die Nummer der Schwester bzw. der Zeile des Dienstplans ist und j die Nummer des Tags bzw. der Spalte des Dienstplans. Damit lassen sich die Variablen dieser Modellierung folgendermaßen als Liste aufschreiben: `[V11,V12,...,V`st`]`.

3.3.3 Constraints

Im folgenden sehen wir, wie bei dieser Modellierung des Dienstplanproblems als CSP die Constraints aussehen. Dabei werden nicht alle in Abschnitt 2 definierten Bedingungen als Constraints formuliert, sondern nur exemplarisch dargestellt, wie das funktioniert.

Belegungssoll der Station

Eine Bedingung für einen korrekten Dienstplan ist das Belegungssoll der Station. Die Belegung einer Station mit Schwestern ist für jede Schicht nach unten und oben begrenzt. `Min1` sei die Mindestbelegung der Station und `Max1` die Höchstbelegung der Station in der Frühschicht. Entsprechend seien `Min2`, `Max2`, `Min3` und `Max3` die Unter- und Obergrenzen für Spät- und Nachtschicht. Damit darf ein korrekter Dienstplan in jeder Spalte nicht weniger als `Min1` mal und nicht mehr als `Max1` mal die 1 als Eintrag haben, nicht weniger als `Min2` und nicht mehr als `Max2` mal die 2 usw.

Wie oben sei s die Anzahl der Schwestern, t die der Tage und `V`ij die Variable für Zeile i und Spalte j. Dann muss für alle $1 \leq j \leq t$ und alle $k \in \{1, 2, 3\}$ folgender Constraint formuliert werden: `cardinality(Min`k`,Max`k`,[V1`j` ?= `k`,V2`j` ?= `k`,...,V`sj` ?= `k`])`

Monatliche Arbeitszeiten der Schwestern

Eine andere Bedingung, die ein korrekter Dienstplan erfüllen muss, ist, dass die monatliche Arbeitszeit jeder Schwester eingehalten wird. In Schichten ausgedrückt heißt das, dass für jede Schwester eine Untergrenze und eine Obergrenze für die Anzahl der Schichten existiert, die sie im Planungszeitraum zu arbeiten hat. Das ist die Anzahl der Früh-, Spät- und Nachtschichten zusammen. Einfacher lässt sich das durch die Anzahl der Freischichten ausdrücken.

Sei wieder s die Anzahl der Schwestern, t die der Tage und Vij die Variable der Zeile i und der Spalte j. Sei für jede Schwester $1 \leq i \leq s$ die Untergrenze für die Arbeitsschichten mit Min$i$ gegeben und die Obergrenze mit Max$i$. Dann lässt sich die Arbeitszeitbedingung für jedes i wie folgt formulieren: `cardinality(`t`-Max`i`,`t`-Min`i`,[V`i`1 ?= 0,V`i`2 ?= 0,...,V`it` ?= 0])`

Damit wird die Anzahl der Freischichten für jede Schwester i begrenzt: nach unten mit t-Max$i$, das ist die Anzahl aller Tage minus der Obergrenze für die Arbeitsschichten, und nach oben mit t-Min$i$, der Anzahl aller Tage minus der Untergrenze für die Arbeitsschichten.

11-Stunden-Regel

Die 11-Stunden-Regel besagt, dass eine Schwester zwischen zwei Arbeitsschichten mindestens 11 Stunden frei haben muss (Abschnitt 2.1.1). Dadurch darf für eine Schwester an aufeinanderfolgenden Tagen auf eine Spätschicht keine Frühschicht und auf eine Nachtschicht nur entweder eine Früh- oder eine Spätschicht folgen.

Sei wieder s die Anzahl der Schwestern, t die der Tage und Vij die Variable für Zeile i und Spalte j. Die 11-Stunden-Regel lässt sich durch Bedingungen folgender Art ausdrücken: wenn Vij mit einem bestimmten Wert belegt ist, dann muss die Belegung von V$i(j+1)$ eine bestimmte Bedingung erfüllen. Das kann mit dem `domain_if`-Constraint formuliert werden. Wir definieren also für jedes $1 \leq i \leq s$ und jedes $1 \leq j < t$:
`domain_if(V`ij` ?= 2, V`$i(j+1)$` ?\= 1)` und
`domain_if(V`ij` ?= 3, V`$i(j+1)$` in [0,3])`.

Freiwünsche

Freiwünsche, sofern sie als harte Wünsche betrachtet werden, führen zu unmittelbaren Variablenbelegungen. Ein Freiwunsch (z.B. Urlaub) der Schwester i am Tag j wird dann durch folgenden Constraint ausgedrückt: `V`ij` ?= 0`.

Weiche Wünsche können wir nicht direkt als Constraints formulieren, da unser Constraint-Löser keine weichen Constraints kennt. Weiche Wünsche können wir nur zur Optimierung von Dienstplänen benutzen.

3.3.4 Bewertung dieser Modellierung

Diese Modellierung ist schön klar und die Constraints sind einigermaßen einfach, allerdings ist der Suchraum viel zu groß. Selbst wie hier, bei der Verwendung eines Constraintlösers, der ja den Suchraum von vornherein und während der Suche nach einer Lösung einschränkt, können reale Dienstplanungsprobleme mit diesem Verfahren nicht in angemessener Zeit gelöst werden. Diese Modellierung ist vielmehr der Ausgangspunkt für andere, bessere Modellierungen und darauf aufbauende Lösungsverfahren.

3.4 Andere Modellierungen

Die oben dargestellte Modellierung ist freilich nicht die einzig mögliche. In diesem Abschnitt sehen wir zwei andere Modellierungen: eine, die die Variablen reduziert und eine, die die Wertebereiche reduziert. Diese wurden im Rahmen dieser Arbeit untersucht.

3.4.1 Reduktion der Variablen

Eine Möglichkeit, die Komplexität des Problems zu verringern, ist die Reduktion der Variablen. Bei gleichen Wertebereichen verringert das unmittelbar den Lösungsraum.

Eine Idee war nun, nicht für jede Zelle des Dienstplans eine Variable zu definieren und diese mit der Nummer einer Schicht zu belegen, sondern für jeden Tag des Planungszeitraums und für jede Schicht und für jede in dieser Schicht benötigte Schwester eine Variable zu haben, die mit der Nummer einer Schwester belegt wird. Damit wird für jeden Tag und jede Schicht von vornherein eine Anzahl an Schwestern festgelegt, die dieser zugeordnet werden sollen, anders als oben, wo dafür Unter- und Obergrenzen definiert waren.

Jede der Variablen wird mit der Nummer einer Schwester belegt. Die Bedingung, dass eine Schwester höchstens eine Schicht pro Tag arbeiten kann, führt zu der Einschränkung, dass alle Variablen eines Tages mit unterschiedlichen Werten belegt werden muss. Außerdem spielt die Reihenfolge der Werte der Variablen in einer Schicht keine Rolle. Die Belegung einer Variable mit der Nummer einer Schwester bedeutet ja, dass diese Schwester in der entsprechenden Schicht arbeiten muss. Für dieselbe Schicht existieren aber noch weitere Variablen, da jede Schicht mit mehr als einer Schwester besetzt sein muss.

Für den typischen Fall einer Station mit 20 Schwestern seien in der Frühschicht 3, in der Spätschicht 3 und in der Nachtschicht 2 nötig. Für jeden Tag benötigen wir also 8 Variablen, deren Werte aus den 20 Nummern der Schwestern gewählt werden. Nachdem die Variablen eines Tages alle verschiedene Werte annehmen müssen und die Reihenfolge der Werte innerhalb der drei Gruppen von Variablen keine Rolle spielen, gibt es für die Belegung der Variablen eines Tages $20!/(3!3!2!12!) = 70543200$ (siehe [Kre91]). Damit hat der Lösungsraum die Komplexität

$$70543200^{30} \approx 10^{235}$$

und damit doch eine erheblich kleinere als die vorige Modellierung.

Allerdings sind die Constraints für diese Modellierung extrem kompliziert. Obige Bedingung, dass die Belegung aller Variablen einer Spalte paarweise verschieden sein müssen, lässt sich noch einfach mit dem `all_distinct`-Constraint formulieren. Auch die Tatsache, dass die Reihenfolge der Werte der Variablen einer Schicht egal ist, kann durch einen einfachen Constraint ausgedrückt werden, der verhindert, dass verschiedene Belegungen mit identischer Bedeutung generiert werden: indem man fordert, dass die Werte geordnet sein müssen, also z.B. aufsteigend. Alles andere aber ist aufwendig. Schon die Formulierung der Belegungsgrenzen pro Schwester, also die Grenzen für die Anzahl der Schichten, die einer Schwester zugeordnet werden, lassen sich mit den gegebenen Constraints nicht formulieren, so dass diese Bedingung nur für eine vollständige Belegung getestet werden kann. Das ist aber wieder ineffizientes *generate and test* (Abschnitt 3.1.3).

Eine Erweiterung dieser Modellierung um die Möglichkeit, die Anzahl der Schwestern in einer Schicht variabel zu halten und nicht von vornherein vorzugeben, was im Übrigen auch häufig verhindert, dass überhaupt eine Lösung für das Problem gefunden wird oder sogar gefunden werden kann, müsste während der Lösung des Problems dem Problem Variablen hinzufügen oder entfernen. Das ist technisch zwar vorstellbar, aber erheblich aufwendig und für die Suche nach einer Lösung sicherlich ineffizient.

3.4.2 Boolesche Wertebereiche

Eine andere Möglichkeit, den Suchraum zu verkleinern ist die Reduktion der Wertebereiche der Variablen. Hier war die erste Idee folgende.

Anstatt für jede Zelle des Dienstplans eine Variable mit Wertebereich $\{0, 1, 2, 3\}$ verwenden wir für jede Zelle drei Variablen mit booleschem Wertebereich $\{0, 1\}$. Jede Variable steht für eine Schicht. Wird die Variable mit 1 belegt, dann muss die entsprechende Schwester am entsprechenden Tag in der zugeordneten Schicht arbeiten. Sind alle drei Variablen einer Zelle mit 0 belegt, bedeutet das Freischicht.

Sofort offensichtlich ist, dass diese Modellierung den Lösungsraum gegenüber der ersten nicht einschränkt. Tatsächlich gibt es für unseren typischen Fall mit 30 Tagen und 20 Schwestern $3 * 20 * 30 = 1800$ Variablen und damit $2^{1800} \approx 10^{542}$ verschiedene Belegungen. Indem aber höchstens eine der drei Variablen einer Zelle mit 1 und alle anderen mit 0 belegt werden, gibt es für die drei Variablen vier verschiedene Belegungen. Damit ist der Suchraum dieses Problems identisch mit dem des ersten, und unser Ziel, den Suchraum einzuschränken, hiermit nicht erreicht.

Diese Modellierung aber zeigt uns eine schön einfache Art, für die Rahmenbedingungen Constraints zu formulieren. Die Belegungsgrenzen pro Tag in den einzelnen Schichten zum Beispiel können durch Unter- und Obergrenze für die Summe über die entsprechenden Variablen einer Spalte ausgedrückt werden.

Sei wieder s die Anzahl der Schwestern, t die der Tage und `V`ijk die Variable für Zeile i und Spalte j und Schicht $k \in \{1, 2, 3\}$. Wie bisher stehe die 1 für *Frühschicht*, die 2 für *Spätschicht* und die 3 für *Nachtschicht*. Außerdem sei `Min`k die Mindestbelegung der Station und `Max`k die Höchstbelegung der Station in der Schicht k. Dann formulieren wir für jedes $1 \leq j \leq t$ und jedes $k \in \{1, 2, 3\}$ die Constraints
`V1`jk `+ V2`jk `+ ...  + V`sjk `?>= Min`k und
`V1`jk `+ V2`jk `+ ...  + V`sjk `?=< Max`k.

Diese Modellierung zusammen mit der Betrachtung der händischen Dienstplanung führte dann zur Formulierung der inkrementellen Lösung (Kapitel 4).

Kapitel 4

Inkrementelle Lösung

Dieser Abschnitt beschreibt zuerst die heute übliche Dienstplanung von Hand und dann unsere darauf aufbauende teilautomatische Dienstplangenerierung: die inkrementelle Erzeugung in mehreren Phasen, die Belegung in Mustern und die Suche nach optimalen Lösungen. Zum Schluss folgen einige Bemerkungen zur Interaktivität.

4.1 Dienstplanung von Hand

Aufgrund des immens großen Suchraums wird ein Dienstplan üblicherweise von Hand in zwei Phasen erzeugt. In der ersten Phase hat man für die Belegung der Zellen noch alle Freiheiten. Deshalb wird da die komplizierteste Belegung (die an die meisten Bedingungen geknüpft ist) erledigt: die Belegung der freien Tage oder der „Freischichten". Diese sind an viele Constraints gebunden: sie bestimmen, wieviele Tage eine Schwester im Belegungszeitraum zu arbeiten hat, wieviele Schwestern insgesamt der Station jeden Tag zur Verfügung stehen und nicht zuletzt sind dabei die meisten Wünsche der Schwestern an den Dienstplan zu berücksichtigen: die Freiwünsche.

Eng mit den Freischichten verknüpft sind die Nachtschichten. Die 11-Stunden-Regel erzwingt ja für die Belegung der Schichten einer Schwester, dass am Tag nach einer Nachtschicht nur entweder eine Nachtschicht oder aber eine Freischicht kommen darf[1]

Deshalb werden bei der händischen Dienstplanerzeugung die Frei- und die Nachtschichten in einer Phase, der ersten, erledigt. In der zweiten Phase werden dann Früh- und Spätschichten auf die noch nicht zugeordneten Zellen des Dienstplans verteilt.

Der offensichtliche Vorteil der Erzeugung in zwei Phasen gegenüber der in einer Phase ist die Reduktion der Komplexität: pro Phase müssen weni-

[1] Die 11-Stunden-Regel gilt zwar auch für die anderen Schichten, doch knüpft sie nur die Nachtschichten (außer an die Nachtschichten selbst) an die Freischichten. Auf alle anderen Schichten dürfen auch Nicht-Freischichten folgen.

ger Constraints und vor allem weniger mögliche Belegungen berücksichtigt werden. In der ersten Phase haben wir für jede Zelle drei Möglichkeiten: Freischicht, Nachtschicht oder *unbestimmt*. In der zweiten werden die noch unbestimmten Zellen mit entweder Frühschicht oder Spätschicht belegt. Die Größe des Lösungsraums ist also (siehe auch Abschnitt 3.2)

$$C = 3^{600} + 2^{600} \approx 3^{600} \approx 10^{286}$$

Der Suchraum ist natürlich viel kleiner. Selbst im schlechtesten Fall probiert ein menschlicher Planer niemals 10^{286} Möglichkeiten, sondern beschränkt sich auf einige wenige. Betrachtet man den Prozess des händischen Planens genauer, so fällt auf, dass ein Mensch selten einzelne Zellen des Dienstplans belegt, sondern wenn möglich ganze Folgen von gleichen Einträgen macht. Man könnte sagen, der menschliche Planer sucht *Muster* im teilweise fertigen Plan und füllt diese aus.

4.2 Phasenweise Generierung

Die Betrachtung der händischen Dienstplanung führte zu folgendem mehrphasigen Verfahren. Dabei sollten alle Phasen möglichst gleich funktionieren, um dieses Prinzip leichter verallgemeinern zu können. Außerdem wollten wir ja den Suchraum einschränken (Abschnitt 3.2).

Die Idee ist, die Anzahl der Phasen soweit zu erhöhen, dass in jeder Phase für jede Zelle nur die minimale Entscheidung zwischen zwei Möglichkeiten bleibt und trotzdem nach allen Phasen ein vollständiger Dienstplan generiert ist. Dazu müssen wir nur die erste Phase der händischen Dienstplanung in zwei teilen. Wir erhalten drei Phasen:

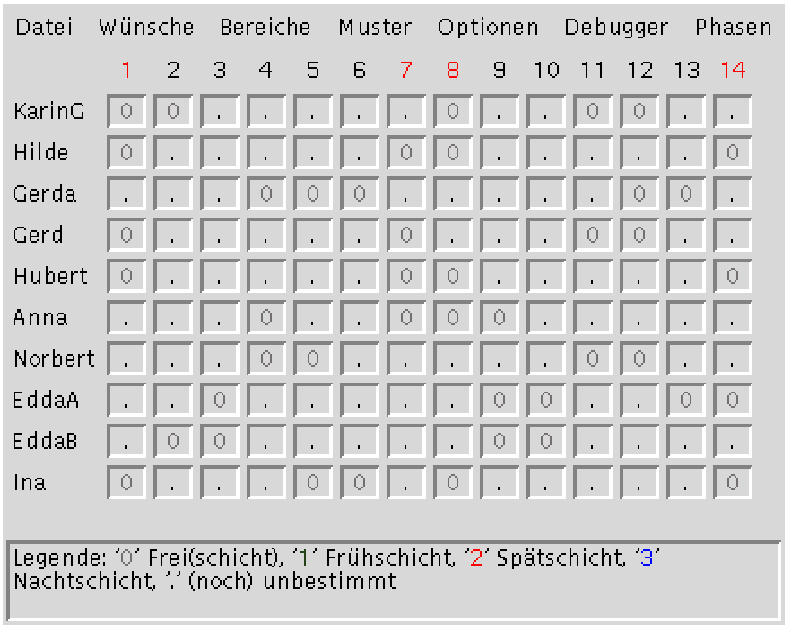
1. Phase Verteilung der Freischichten auf die Zellen des Dienstplans.

2. Phase Verteilung der Nachtschichten auf die noch nicht belegten Zellen.

3. Phase Verteilung der Früh- und der Spätschichten auf die restlichen Zellen.

In jeder Phase wird jede Variable an einen Wert aus dem booleschen Wertebereich $\{0, 1\}$ gebunden. Je nach Phase haben die Werte Null und Eins verschiedene Bedeutung. Die Belegung einer Variable mit einem Wert führt unter Umständen zu einem Eintrag an der entsprechenden Stelle des Dienstplans. Ein vollständiger Dienstplan ergibt sich aus allen drei Phasen.

Wenn einer Variable in der ersten Phase der Wert Null zugeordnet wird, tragen wir in die entsprechende Zelle des Dienstplans eine Freischicht ein. Die Zellen, deren Variablen an die Eins gebunden sind, bleiben unentschieden. Abbildung 4.1 zeigt die Teillösung eines Dienstplans nach der ersten Phase. Eine „Teillösung“ ist immer ein zwar vollständig belegter Dienstplan, also nicht ein während der Berechnung einer Phase nur teilweise belegter, der aber unbestimmte Zellen enthält. Eine Teillösung ist also zwar das Resultat einer vollständigen Phase, aber nicht der letzten Phase.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
KarinG	0	0	.	.	.	.	.	0	.	.	0	0	.	.
Hilde	0	.	.	.	.	.	0	0	.	.	.	.	.	0
Gerda	.	.	.	0	0	0	.	.	.	.	.	0	0	.
Gerd	0	.	.	.	.	.	0	.	.	.	0	0	.	.
Hubert	0	.	.	.	.	.	0	0	.	.	.	.	.	0
Anna	.	.	.	0	.	.	0	0	0	.	.	.	.	.
Norbert	.	.	.	0	0	.	.	.	.	.	0	0	.	.
EddaA	.	.	0	.	.	.	.	.	0	0	.	.	0	0
EddaB	.	0	0	.	.	.	.	.	0	0	.	.	.	.
Ina	0	.	.	.	0	0	.	0	.	.	.	.	.	0

Abbildung 4.1: Dienstplan nach der ersten Phase

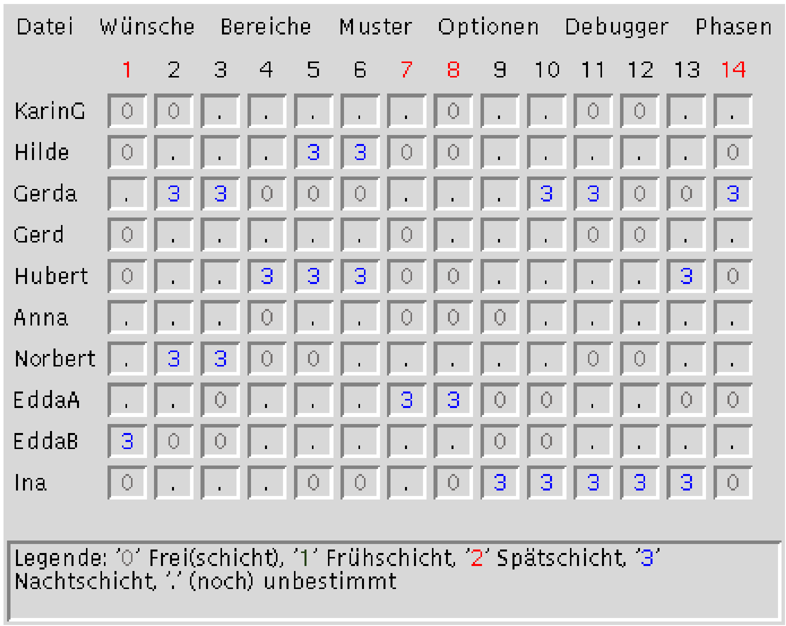
Die zweite Phase behandelt ausschließlich die unentschiedenen Zellen: erhält eine Variable den Wert Eins, wird in die Zelle eine Nachtschicht eingetragen, der Rest bleibt weiterhin unentschieden. In Abbildung 4.2 sehen wir ein Beispiel eines Dienstplans nach der zweiten Phase.

In der dritten Phase werden die noch unentschiedenen Zellen vollständig mit Frühschicht oder Spätschicht belegt, je nachdem, ob die entsprechende Variable an Eins bzw. Null gebunden ist. Nach der dritten Phase ist der Dienstplan komplett (siehe Abbildung 2.1). Abbildung 4.3 veranschaulicht die Belegung der Variablen in den drei Phasen und die damit verbundenen Einträge in den Dienstplan.

Dieses Verfahren reduziert den Suchraum auf

$$C = 2^{600} + 2^{600} + 2^{600} \approx 10^{181}$$

Diese Komplexität setzt voraus, dass die Phasen unabhängig sind. Tatsächlich gilt das für den automatischen Teil der Planung mit INTERDIP auch: Belegungen, die in früheren Phasen vorgenommen wurden, werden in späteren nicht mehr rückgängig gemacht. Allerdings kann es sein, dass der Benutzer fürhere Phasen neu berechnet. Die Komplexität ist also nur im Zusammenhang mit der Bedienung durch den Benutzer feststellbar. Genauer



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
KarinG	0	0	.	.	.	.	.	0	.	.	0	0	.	.
Hilde	0	.	.	.	3	3	0	0	.	.	.	.	.	0
Gerda	.	3	3	0	0	0	.	.	.	3	3	0	0	3
Gerd	0	.	.	.	.	.	0	.	.	.	0	0	.	.
Hubert	0	.	.	3	3	3	0	0	.	.	.	.	3	0
Anna	.	.	.	0	.	.	0	0	0	.	.	.	.	.
Norbert	.	3	3	0	0	.	.	.	.	.	0	0	.	.
EddaA	.	.	0	.	.	.	3	3	0	0	.	.	0	0
EddaB	3	0	0	.	.	.	.	.	0	0	.	.	.	.
Ina	0	.	.	.	0	0	.	0	3	3	3	3	3	0

Abbildung 4.2: Dienstplan nach der zweiten Phase

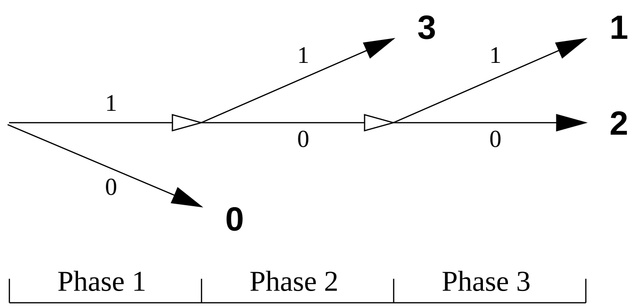


Abbildung 4.3: Belegung der Zellen in drei Phasen

betrachtet wird dieses Problem am Ende dieser Arbeit: in der Zusammenfassung, Abschnitt 7.2.

4.3 Belegung in Mustern

Dienstpläne werden mit INTERDIP also in mehreren Phasen erzeugt. Innerhalb einer Phase aber gibt es immer noch sehr viele Möglichkeiten, den Dienstplan zu belegen, der Lösungsraum einer Phase ist also sehr groß. Der Versuch des menschlichen Dienstplaners, bei der Belegung des Plans nicht einzelne Zellen zu belegen, sondern auf einmal ganze Folgen von Einträgen zu machen, und das nach bestimmten Mustern, reduziert den Suchraum weiter.

Das hier eingesetzte Belegungsverfahren verwendet phasenspezifische *Muster*, die vom Benutzer definiert werden. Wie wir oben gesehen haben, werden die Variablen in jeder Phase mit 0 oder 1 belegt und entsprechend Schichten in den Dienstplan eingetragen. Muster sind dann *sinnvolle* Kombinationen von *Dienstplan*-Einträgen, wobei eine Kombination für aufeinanderfolgende Tage steht. Zum Beispiel halten wir folgende Kombination für sinnvoll: fünf Tage arbeiten, zwei Tage frei. Das entsprechende Muster für die erste Phase, in der die Arbeitstage ermittelt werden, ist: (unbestimmt,unbestimmt,unbestimmt,unbestimmt,unbestimmt,0,0). Unter der Annahme, dass es besser ist, an drei aufeinanderfolgenden Tagen in derselben Schicht zu arbeiten als in unterschiedlichen, formulieren wir für die zweite Phase und damit für die Nachtschichten: (3,3,3). Jede Phase hat einen eigenen Satz von Mustern. Die Muster einer Phase haben eine Reihenfolge, nach der sie benutzt werden: zuerst solche, die ein *gutes* Ergebnis erzeugen– hier sollen im obigen Sinne sinnvolle Kombinationen stehen – und am Schluss triviale einelementige, die notwendig sind um Lösungen zu generieren, wenn sie existieren. Die Variablen und damit die Zellen des Dienstplans werden nur nach den vorgegebenen Mustern belegt.

Indem mehrere Variable auf einmal belegt werden, wird der Suchraum weiter eingeschränkt. Werden im Schnitt drei Variable auf einmal belegt, und das war in unseren Tests tatsächlich der Fall, hat der Suchraum in jeder Phase die Größe $2^{600/3}$ und insgesamt:

$$C = 3 * 2^{200} \approx 10^{61}$$

Trotz der genannten Verbesserungen ist der Suchraum immer noch zu groß. Diese theoretische Komplexität wird in der Praxis weiter verringert, da bei konkreten Problemen viele Constraints berücksichtigt werden, die den Suchraum bei Verwendung eines Constraintlösers unmittelbar einschränken. Bei einer typischen Anzahl von Constraints generiert und optimiert INTERDIP eine Lösung im Allgemeinen binnen einiger Minuten, was bei den oben genannten größeren Suchräumen kaum möglich ist. Insofern ermöglicht erst die angegebene Reduzierung des Suchraums das Finden einer Lösung innerhalb eines brauchbaren Zeitraums von einigen Minuten.

4.4 Optimale Dienstpläne

Was ist ein optimaler Dienstplan? Wie wir oben gesehen haben, beeinflusst die Auswahl der Muster die Güte eines Dienstplanes. Da im Allgemeinen nicht alle Constraints erfüllt werden können, müssen wir einen Teil der Constraints als weiche Constraints behandeln. Die Qualität eines Dienstplans hängt also auch damit zusammen, welche weichen Constraints erfüllt sind.

Der Constraintlöser von IF/Prolog kann von Hause aus keine weichen Constraints behandeln. Jede Variablenbelegung erfüllt alle Constraints, die an ihn gestellt werden. Deshalb formulieren wir nur die harten Constraints als IF/Prolog-Constraints und weisen jedem weichen Constraint des Problems fiktive *Kosten* zu, die *entstehen*, wenn das Constraint verletzt wird. Kosten entstehen dabei für jede Schwester getrennt. Die Kosten werden dann für folgendes Optimierungsverfahren verwendet, das man auch *branch-and-bound* nennt [CS93].

In jeder Phase wird zuerst eine Variablenbelegung für alle Zellen generiert. Diese Lösung L wird zu einer Lösung L' optimiert, indem alle Variablenbindungen rückgängig gemacht werden und dieselbe Belegung, die zu L geführt hat, erneut versucht wird, mit dem Unterschied, dass alle Kosten von L' kleiner sein müssen als das Maximum der Kosten von L (*minimize-maximum-branch-and-bound*). INTERDIP optimiert dann L' genauso weiter zu L'' und so fort. Dieser Prozess und damit diese Phase ist zu Ende, wenn entweder eine vom Benutzer vorgegebene Zeitschranke erreicht ist oder der Benutzer selbst die Berechnung abbricht oder eine optimale Lösung gefunden ist. Eine Lösung ist optimal, wenn sie nicht mehr verbessert werden kann, also kein L' gefunden werden kann, deren Kosten alle unter dem Maximum von L liegen.

Kosten entstehen also getrennt für jede Schwester und das Maximum dieser Kosten wird minimiert. INTERDIP versucht zu erreichen, dass keine Schwester besonders schlecht wegkommt und damit zum Beispiel die Anzahl der unerfüllten Wünsche gleichmäßig über alle Schwestern verteilt wird. Das führt zu einem Dienstplan, der allgemein für gerecht gehalten wird.

INTERDIP besitzt damit zwei Techniken, optimale Lösungen zu erzeugen:

- Art und Reihenfolge der Muster für die Belegung
- Verletzungen weicher Constraints werden minimiert

4.5 Interaktivität

Teil des pragmatischen Ansatzes ist die Interaktivität. INTERDIP ist nicht dafür gedacht, aufgrund vorgegebener Rahmenbedingungen selbständig einen optimalen Dienstplan zu erzeugen. Die besten und schnellsten Resultate erhält der Benutzer in Interaktion mit INTERDIP.

Dafür stehen ihm zum einen alle Rahmenbedingungen zur grafischen Erfassung zur Verfügung. Der Prototyp implementiert diese vollständig. Außerdem kann der Benutzer mit einem Debugger in den automatischen Planungsprozess eingreifen und mit einem weiteren Tool generierte Pläne bearbeiten und deren Korrektheit, das heißt Konsistenz bezüglich aller Constraints, überprüfen. Inkonsistenzen werden dem Benutzer so genau wie möglich angezeigt. Das heißt, es wird die Art der Inkonsistenz angezeigt und die Stelle, an der sie vorliegt.

Auch durch Veränderung der Rahmenbedingungen nach erfolgter Planung und Neuanstoßen der entsprechenden Phasen kann der Benutzer einen besseren Plan bekommen.

Detailliert sind alle diese Möglichkeiten und deren Verwendung im Handbuch (Kapitel 6) beschrieben.

Kapitel 5

Realisierung von INTERDIP

Dieses Kapitel beschreibt die Realisierung von INTERDIP, der prototypischen Implementierung der inkrementellen Dienstplanung, wie sie in Kapitel 4 beschrieben ist. Dazu wird zuerst IF/Prolog kurz dargestellt, zusammen mit für diese Arbeit nötigen Zusätzen, dann wird die Spezifikation des Problems durch Prolog-Prädikate beschrieben. Darauf folgt die Darstellung der Implementierung des mehrphasigen Dienstplaners. Anschließend wird die Implementierung der Benutzerschnittstelle und der interaktiven Möglichkeiten dargestellt. Am Schluss stehen Bemerkungen zur Realisierung möglicher Erweiterungen des Prototypen.

5.1 IF/Prolog

IF/Prolog ist das Prolog-Entwicklungssystem der Siemens Nixdorf Informationssysteme. IF/Prolog ist ISO-konform [Sie96b, ISO94], besitzt aber über den ISO-Standard hinaus etliche Erweiterungen, deren für diese Arbeit wichtigste das Constraints-Paket [Sie96a] mit Constraint-Lösern für die Constraint-Systeme Lineare Gleichungen, Coroutinen, FD-Constraints und Boolesche Constraints.

5.1.1 Zusätzliche Prädikate

IF/Prolog definiert von Haus aus viele Prolog-Standardprädikate. Das liegt schon daran, dass IF/Prolog ISO-konform ist. Standardprädikate seien die Prädikate, die Sterling und Shapiro in ihrem Prolog-Standardwerk [SS94] beschrieben haben und deshalb hier nicht dargestellt werden. Im folgenden Abschnitt werden Nichtstandard-Prädikate definiert, die für die Realisierung benötigt wurden. Das sind vor allem Rekursions-Metaprädikate, Prädikate also, mit denen für andere Prädikate Rekursion realisiert werden kann, ohne dass diese aufwendig hingeschrieben werden muss. Die Prädikate sind nicht

Teil von IF/Prolog sondern wurden speziell für diese Arbeit entworfen. Deshalb ist auch die Implementierung angegeben.

`transpose(M,Mt)` ⟵ `M` ist eine Matrix (eine Liste von gleich langen Listen) und `Mt` ist die transponierte dazu, z.B. gilt `transpose([[1,2,3],[4,5,6]],[[1,4],[2,5],[3,6]])`.

```
transpose([[]|_],[]) :- !.
transpose(M,Mt) :-
        not var(M),
        transpose(M,MCol,M_),
        transpose(M_,Mt_),
        Mt=[MCol|Mt_].

transpose([],[],[]) :- !.
transpose([[A|As]|R],[A|Bs],[As|R_]) :-
        transpose(R,Bs,R_).
```

`rec_doall(List,Elem,Goal)` ⟵ `Goal` gilt für alle Elemente `Elem` der Liste `List`. Das ist eine Art rekursives `doall` auf einer Liste. Dabei ist `Elem` in der Regel Teilausdruck von `Goal` und dieser wird sukzessive durch alle Elemente der Liste `List` ersetzt und damit `Goal` aufgerufen.

```
rec_doall([],_,_).
rec_doall([L|Ls],E,Goal) :-
        copy_term((E,Goal),(L,Goal_)),
        Goal_,
        rec_doall(Ls,E,Goal).
```

`rec_doall_i(List,Elem,Index=Start,Goal)` ⟵ wie bei `rec_doall/3` gilt `Goal` für alle Elemente `Elem` der Liste `List`, und `Index` ist die Rekursionstiefe bzw. die Nummer des Elements in der Liste. Die Zählung dafür beginnt bei `Start`.

```
rec_doall_i([],_,_,_).
rec_doall_i([L|Ls],E,I=S,Goal) :-
        copy_term((E,I,Goal),(L,S,Goal_)),
        Goal_,
        S_ is S+1,
        rec_doall_i(Ls,E,I=S_,Goal).
```

`rec_doall_0(List,Elem,Index,Goal)` ⟵
`rec_doall_i(List,Elem,Index = 0,Goal).`

`rec_doall_1(List,Elem,Index,Goal)` ⟵
    `rec_doall_i(List,Elem,Index = 1,Goal).`

`mapper(ListList,VarList,Goal)` ⟵ `ListList` ist eine Liste gleichmächtiger Listen und `VarList` eine Liste von Variablen, für jede Liste aus `ListList` eine Variable, und `Goal` gilt für alle entsprechenden Elemente (`VarList`) der Listen aus `ListList`. Zur Anwendung siehe auch Abschnitt 5.3.4.

```
%mapper(ListList,VarList,Goal) :-
mapper(LL,VL,G) :-
        transpose(LL,LLt),
        rec_doall(LLt,VL,G).
```

`mapperi(ListList,VarList,Index=Start,Goal)` ⟵ wie bei `mapper/3` gilt `Goal` für alle Elemente `VarList` der Listen `ListList` und `Index` ist die Nummer des Elements und das erste Element hat die Nummer `Start`.

```
%mapperi(ListList,VarList,Index=Start,Goal) :-
mapperi(LL,VL,I=S,G) :-
        transpose(LL,LLt),
        rec_doall_i(LLt,VL,I=S,G).
```

`mapper0(LL,VL,I,G)` ⟵ `mapperi(LL,VL,I=0,G).`

`mapper1(LL,VL,I,G)` ⟵ `mapperi(LL,VL,I=1,G).`

5.2 Spezifikation

Folgende Prädikate stellen die Spezifikation des Problems dar, definieren also die Rahmenbedingungen des zu erstellenden Dienstplans.

`schwestern(N)` ⟵ `N` ist die Anzahl der Schwestern, für die ein Dienstplan erstellt werden soll. Das ist die Anzahl der Zeilen des Dienstplans.

`tage(N)` ⟵ `N` ist die Anzahl der Tage des Planungszeitraums. Das ist die Anzahl der Spalten des Dienstplans.

`zeitschranke(Z)` ⟵ `Z` ist die Zeitschranke (in Sekunden), nach der die Berechnung einer Phase automatisch abgebrochen wird.

`muster(Phase,Muster,Kosten)` ⟵ `Muster` ist ein Muster, das in Phase `Phase` zur Belegung des Dienstplans verwendet wird. Wird `Muster` tatsächlich angewendet, entstehen dafür `Kosten` Kosten. `Muster` ist eine Liste von aufeinanderfolgenden Dienstplaneinträgen.

`belegung(Phase,TagRest,Von,Bis)` ⟵ in Phase `Phase` gilt für `TagRest` die Minimalbelegung `Min` und die Maximalbelegung `Max`. `TagRest` hat für die erste Phase (`Phase == 1`) eine andere Bedeutung als für die restlichen Phasen (`Phase > 1`):

`Phase == 1 : TagRest` bezeichnet den Tag, für den die Belegungsgrenzen gelten sollen. Damit werden für die erste Phase die Belegungsgrenzen für jeden Tag einzeln spezifiziert.

`Phase > 1 : TagRest` bezeichnet den *Rest* an Schwestern, der für die Belegung in dieser Phase noch zur Verfügung steht. Das ist die Anzahl der Schwestern, die an einem Tag den Eintrag *unbestimmt* haben.

`bereich(Phase,Schwester,Von,Bis)` ⟵ `Von` ist die Untergrenze und `Bis` die Obergrenze zur Zuweisung von Schichten für die Schwester mit der Nummer `Schwester` in der Phase `Phase`. Dadurch wird für die Phase 1 die Arbeitszeit jeder Schwester im Planungszeitraum, das ist die Anzahl der Schichten, die die Schwester mindestens und höchstens zu arbeiten hat, definiert. Für die Phase 2 wird die Anzahl der Nachtischten für jede Schwester begrenzt und für die Phase 3 die der Frühschichten (siehe auch Kapitel 6).

`wuensche(Farbe,Kosten,Liste)` ⟵ `Liste` enthält die Wünsche der Farbe `Farbe`. `Farbe` ist entweder `rot`, `schwarz` oder `weiß`. Wird ein Wunsch verletzt, entstehen dafür `Kosten` Kosten. Ist `Kosten == 10000`, werden alle diese Wünsche als harte Constraints behandelt. `Liste` ist eine Liste von Listen, die alle die Wünsche einer Zeile des Dienstplans beschreiben.

`einschraenkung(Name,Schalter)` ⟵ Die Einschränkung `Name` ist eingeschaltet (`Schalter == ein`) bzw. ausgeschaltet (`Schalter == aus`). Globale Einschränkungen definieren z.B. die 11-Stunden-Regel. Siehe auch Abschnitt 5.3.4 für konkret definierte Einschränkungen.

5.3 Der mehrphasige Dienstplaner

Wie in Kapitel 4 dargestellt, generiert INTERDIP einen Dienstplan in mehreren Phasen. Wie das realisiert ist, wird im folgenden dargestellt. Dabei wird die Realisierung der Benutzerschnittstelle und was damit zusammenhängt nach Möglichkeit weggelassen. Darauf wird im nächsten Abschnitt genauer eingegangen.

Die Grundstruktur des Dienstplaners ist folgende. Nach dem Start von INTERDIP, die Rahmenbedingungen (siehe Abschnitt 5.2) sind geladen, werden zuerst die Variablen erzeugt (Abschnitt 5.3.2). Für alle Phasen und damit im ganzen IF/Prolog-Programm existiert nur ein Satz Domainvariablen. Diese werden in jeder Phase belegt und nach dem Ende der Phase werden alle Variablenbindungen wieder aufgehoben. Sind die Variablen erzeugt, werden (nach Anforderung vom Benutzer) die Phasen berechnet (Abschnitt 5.3.3). Beim Berechnen einer Phase wird zuerst geprüft, ob für die dieser Phase zugrundeliegende Phase (bei Phase n die Phase $n-1$; Phase 0 ist eine „Pseudo"-Phase mit einer trivialen Lösung) schon Lösungen berechnet wurden, von denen dann die beste verwendet wird. Wenn nicht, wird jene Phase zuerst berechnet und evtl. rekursiv die vorhergehende usw. Ist eine Lösung für die vorhergehende Phase gefunden, werden die Constraints formuliert (Abschnitt 5.3.4). Anschließend wird eine optimale Lösung berechnet (Abschnitt 5.3.5). Dazu wird zuerst nach einer Lösung gesucht und diese dann optimiert (siehe Abschnitt 4.4). Jede dadurch gefundene Lösung wird in der Prolog-Faktenbasis gespeichert. Ist die Zeitschranke erreicht, wird die Berechnung abgebrochen und die Belegung aller Domainvariablen rückgängig gemacht: das Prädikat, das die Lösung sucht, scheitert immer.

5.3.1 Hauptprogramm

Das INTERDIP-Hauptprogramm, das also, das nach dem Start von INTERDIP als erstes aufgerufen wird und alle weiteren Aktionen einleitet, ist das Prädikat `program/0` (das ist bei IF/Prolog standardmäßig das Prädikat, das, falls vorhanden, unmittelbar nach dem Start von IF/Prolog aufgerufen wird):

`program` ⟵ „INTERDIP".

```
program :-
        % Tk initialisieren und Datei laden
        tk([file('interdip.tcl')]),

        s erz_ds(DS),

        repeat,
        tk_do_one_event(0), % auf Tk-Ereignis warten
        (current_global(phase) ->
            get_global(phase,Phase),
            phase s Phase s not phase(Phase,DS),
            unset_global(phase)
        ;
            unset_global(phase)
        ),
        (current_global(konsistenz_pruefen) ->
            get_global(konsistenz_pruefen,Plan),
```

```
            konsistenz_pruefen(Plan,DS),
            unset_global(konsistenz_pruefen)
        ;
            unset_global(konsistenz_pruefen)
        ),
        fail.
```

`s(Status,Goal)` ⟵ `Status` ist in der Statuszeile der Benutzerschnittstelle angezeigt und `Goal` gilt. `s/2` ist als rechtsassoziativer Infix-Operator deklariert und bindet gerade stärker als der Komma-Operator und gerade schwächer als der `not`-Operator.

`s(Goal)` ⟵ `Goal` gilt und wird in der Statuszeile der Benutzerschnittstelle angezeigt. `s/1` ist als Präfix-Operator deklariert und bindet gerade stärker als der Komma-Operator und gerade schwächer als der `not`-Operator.

`current_global(Name)` ⟵ es existiert eine globale Variable mit Namen `Name`.

`get_global(Name,Wert)` ⟵ die globale Variable mit Namen `Name` hat den Wert `Wert`.

`unset_global(Name)` ⟵ die globale Variable `Name` ist nicht mehr definiert.

`set_global(Name,Wert)` ⟵ die globale Variable ist jetzt definiert (auch wenn sie noch nicht definiert war) und hat jetzt den Wert `Wert`.

Hier (in `program/0`) wird zuerst der Tk-Teil von INTERDIP initialisiert (siehe Abschnitt 5.4.2) und anschließend wird die Datenstruktur DS erzeugt (siehe unten). Danach wird in einer unendlichen Schleife auf Tcl/Tk-Ereignisse gewartet und diese entsprechend ausgewertet. Wichtig ist an dieser Stelle der Aufruf `phase s Phase s not phase(Phase,DS)`, der stattfindet, wenn von der Benutzerschnittstelle her die Berechnung einer Phase angestoßen wird. In dieser Zeile wird der zweistellige Operator `s` verwendet. Die Zeile ist äquivalent zum folgenden Aufruf: `s(phase,s(Phase,not(phase(VorPhase,DS))))`. Neben den Anzeigen in der Benutzerschnittstelle, wird nur `not(phase(VorPhase,DS))` aufgerufen. `phase(Phase,DS)` berechnet die Phase `Phase` mit den Variablen aus `DS`. `phase/2` scheitert immer (damit die Belegung der Variablen aus DS rückgängig gemacht wird). Deshalb wird es hier negiert aufgerufen (siehe auch Abschnitt 5.3.3).

Der hier nicht beschriebene Rest von `program/0` wird in Abschnitt 5.4.2 erklärt.

5.3.2 Variablen

Ein zentrales Problem der Realisierung ist die Verwaltung der Domain-Variablen. Domain-Variablen sind die Variablen, die das CSP definieren und für die Wertebereiche und Constraints definiert sind. Wir nennen diese oft auch kurz *Variablen*.

In Prolog existieren zwei Techniken der Wiederholung von Anweisungen: Rekursion und *failure-driven-loop*, also die Wiederholung durch Scheitern (*failure*) und Rückverfolgung (*backtracking*)[SS94]. Mit Domain-Variablen kann die zweite Konstruktion nicht verwendet werden. Durch Backtracking werden nämlich nicht nur Variablenbelegungen rückgängig gemacht, sondern auch Constraints, die für Variablen definiert wurden. Mit Domain-Variablen kann also ein Programm nur gewissermassen „vorwärts“ ausgewertet werden, das heißt ohne Backtrackingschritte. Programmiertechnisch bleibt also nur Rekursion.

Der Umgang mit Domainvariablen hat aber noch ein anderes Problem: für Strukturen, die Domainvariablen enthalten, kommt keine Ablage in der Faktenbasis in Frage. Wird nämlich in Prolog ein beliebiger Term in die Faktenbasis geschrieben, und später abgefragt, sind die Variablen des Terms durch neue ersetzt. Damit fallen für Domainvariablen alle Constraints weg. Damit müssen die Domainvariablen des CSP an alle Prädikate, die sie benötigen, übergeben werden. Für INTERDIP habe ich dafür eine große, zum Teil dynamische Datenstruktur entworfen und implementiert. Ich nenne diese Struktur einfach DS. Diese Struktur enthält alle Domainvariablen plus einige notwendige Zusatzinformationen.

`erz_ds(DS)` ⟵ `DS` ist die Datenstruktur DS, die alle Variablen des Problems enthält und diese in verschiedenen Strukturen: S0, S1, S21, S22, S31, S32, SW0, etc. Das Prädikat `erz_ds/1` stützt sich im wesentlichen auf `erz_vars/1`, das die eigentlichen Variablen erzeugt, und macht daraus die Struktur DS.

Für die Formulierung des CSP benötigen wir zunächst für jede Zelle des Dienstplans eine Variable. Das ist sehr einfach:

`erz_vars(Vars)` ⟵ `Vars` ist eine Liste von 3-Tupeln `[Schwester,Tag,Var]`, für jede Schwester und jeden Tag eins; an der dritten Stelle des Tupels steht die Domainvariable.

```
erz_vars(Vs) :-
        findall([S,T,_],(schwester(S),tag(T)),Vs).
```

`for(Start,Counter,End)` ⟵ `Counter` hat den Wert `Start` oder `Start+1` oder ... oder `End`.

`schwester(S)` ⟵ `S` ist die Nummer einer Schwester. Die Schwestern der Station werden dabei einfach durchnummeriert.

```
schwester(S) :-
        schwestern(Schwestern),
        for(1,S,Schwestern).
```

`tag(T)` ⟵ `T` ist ein Tag des Planungszeitraums. Die Tage werden dabei einfach durchnummeriert.

```
tag(T) :-
        tage(Tage),
        for(1,T,Tage).
```

Die Variablen werden mit `erz_vars/1` in einer Liste von Tupeln erzeugt. Jedes Tupel ist eindeutig durch die Nummer der Schwester und dem Tag identifiziert. Damit kann auf jede Domain-Variable eindeutig Bezug genommen werden.

Die Struktur der Tuple-Liste ist aber im Zugriff sehr aufwendig und damit ineffizient. Wollen wir beispielsweise für jede Zeile des Dienstplans einen Constraint formulieren, müssen wir für jede Zeile die entsprechenden Elemente aus der Tupelliste sammeln und auf diese dann den Constraint anwenden. DS enthält deshalb die Domainvariablen in verschiedenen Datenstrukturen. Einmal enthält sie alle Domainvariablen in obiger Tupelliste. Als Beispiel sei diese Liste für zwei Zeilen und drei Spalten gegeben:

```
[[1,1,V11],[1,2,V12],[1,3,V13],[2,1,V21],[2,2,V22],[2,3,V23]]
```

Diese Darstellung nenne ich auch „S1“[1]. Wie diese Bezeichnung verwendet wird, sehen wir später.

Dann enthält sie eine Liste mit allen Domainvariablen als Element:

```
[V11,V12,V13,V21,V22,V23]
```

Diese Struktur heißt „S0“. Die verschiedenen Darstellungen enthalten tatsächlich dieselben Variablen, das heißt, die entsprechenden Variablen werden bei der Erzeugung von DS unifiziert.

Weiterhin enthält DS eine Liste aller Zeilen des Dienstplans, das heißt, eine Liste von Listen, deren Elemente wiederum die Domainvariablen sind („S31“):

```
[[V11,V12,V13],[V21,V22,V23]]
```

[1]Zur Bezeichnung der Strukturen: S ist einfach der Anfangsbuchstabe von *Struktur* und die Zahlen nummerieren die Strukturen, von der einfachsten zur komplexesten: 0, 1, $2x$, $3x$. x ist dabei eine zweite Nummer, die zeilenweise Struktur trägt die 1 und die spaltenweise die 2.

Genauso enthält DS eine Liste aller Spalten des Dienstplans („S32"):

```
[[V11,V21],[V12,V22],[V13,V23]]
```

Schließlich enthält DS zwei indizierte Datenstrukturen. Die eine ist zuerst nach der Zeilennummer und dann nach der Spaltennummer geordnet („S21"):

```
[1-[1-[V11],2-[V12],3-[V13]],2-[1-[V21],2-[V22],3-[V23]]]
```

Die andere ist zuerst nach der Spaltennummer und dann nach der Zeilennummer geordnet („S22"):

```
[1-[1-[V11],2-[V21]],2-[1-[V12],2-[V22]],3-[1-[V13],2-[V23]]]
```

DS enthält darüberhinaus noch andere Strukturen, vor allem alle genannten (S0-S32) nochmal, nur dass anstelle der Domainvariablen die Wünsche für die entsprechende Zelle des Dienstplans stehen (W0-W32) und alle nochmal und zwar für jede Zelle die Domainvariable und die Wünsche (SW0-SW32). Außerdem enthält DS noch eine ganz anders geartete Struktur zur Speicherung der Kosten. Diese ist während der Laufzeit des Programm erweiterbar. Alle diese Strukturen werden später noch genauer erläutert.

5.3.3 Berechnung einer Phase

Dieser Abschnitt beschreibt die Berechnung einer Phase. Dazu dient das Prädikat `phase/2`.

`phase(Phase,DS)` ⟵ Phase `Phase` wurde für die Variablen der Datenstruktur `DS` berechnet (Erläuterungen siehe unten).

```
phase(0,_) :-
        findall(L,
            (schwester(_),findall('.',tag(_),L)),
            NullteLoesung),
        findall(0,schwester(_),NullteLoesungKosten),
        asserta(
            loesung(0,NullteLoesung,NullteLoesungKosten)),
        !,
        fail.

phase(Phase,DS) :-
        Phase > 0,
        VorPhase is Phase-1,

        doall((
```

```
        loesung(Ph,_,_),
        Ph >= Phase,
        retract(loesung(Ph,_,_))
    )),

    ( loesung(VorPhase,_,_) ->
        true
    ;
        phase s VorPhase s not phase(VorPhase,DS) ),
    beste_loesung(VorPhase,Plan,_),

    constraints_eintragen(Phase,Plan,DS),

    doall(belegen(Phase,DS)),

    !,fail.
```

`loesung(Phase,Plan,Kosten)` ⟵ `Plan` ist eine Teillösung (siehe Kapitel 4) der Phase `Phase` mit den Kosten `Kosten`. `Plan` ist eine Liste von Zeilen des Dienstplans (für jede Schwester eine) und jede Zeile ist eine Liste von Einträgen (für jeden Tag einen). Ein Eintrag ist entweder `0` für *Freischicht*, `1` für *Frühschicht*, `2` für *Spätschicht*, `3` für *Nachtschicht* oder `'.'` für *noch unbestimmt*. `Kosten` ist eine Liste von Kosten (das sind Domainvariablen bzw. Zahlen), ein Eintrag pro Schwester (siehe auch Abschnitt 4.4). Dieses Prädikat dient als Fakt der Datenbasis und damit zur Speicherung bereits berechneter Lösungen.

`beste_loesung(Phase,Plan,Kosten)` ⟵ `Plan` ist die beste Lösung der Phase `Phase` mit den Kosten `Kosten`. Ist `Phase` eine (ungebundene) Variable, so liefert `beste_loesung/3` die Lösung der zuletzt berechneten Phase.

Das Prädikat `phase/2` berechnet eine Phase. Dazu werden zwei Paramter übergeben: `Phase`, die Nummer der Phase und `DS`, die Datenstruktur mit den Domainvariablen. Betrachten wir die erste Klausel des Prädikats (`phase(0,_) :- ...`). Hier wird die triviale „Lösung“ der nullten Phase erzeugt und als Fakt `loesung(Phase,Plan,Kosten)` abgespeichert. Die `Phase` ist 0, der `Plan` enthält nur die trivialen Eintragungen `'.'` (das triviale an der nullten Lösung ist ja gerade, dass die Belegung aller Zellen des Dienstplans noch unbestimmt ist) in jeder Zelle und die `Kosten` sind überall 0.

Die eigentliche Berechnung einer Phase, also der nichttrivialen Phasen mit der Nummer größer als Null, ist im zweiten Teil definiert. Hier wird zuerst die Nummer der dieser Phase vorausgehenden Phase bestimmt:

`VorPhase is Phase-1`. Dann werden alle schon existierenden Lösungen dieser Phase (aus vorherigen Berechnungen) gelöscht. Dann wird geprüft, ob zu der dieser Phase vorausgehende Phase (mit der Nummer `VorPhase`) bereits eine Lösung existiert. Wenn nicht, wird `phase/2` rekursiv für `VorPhase` aufgerufen: `phase s VorPhase s not phase(VorPhase,DS)`. In dieser Zeile wird der zweistellige Operator `s` verwendet (Abschnitt 5.3.1. Diese Zeile ist äquivalent zum folgenden Aufruf: `s(phase,s(Phase,not(phase(VorPhase,DS))))`. Neben den Anzeigen in der Benutzerschnittstelle, wird nur `not(phase(VorPhase,DS))` aufgerufen. Warum `phase/2` negiert aufgerufen wird, werden wir gleich noch sehen.

Anschließend werden die Constraints dieser `Phase` bezüglich dem besten bisherigen `Plan` in die Datenstruktur `DS`, d.h. in die darin enthaltenen Domainvariablen, eingetragen. Schließlich werden die Variablen belegt. Das Prädikat `belegen/2` scheitert (fast) immer, dann nämlich, wenn die Berechnung nach abgelaufener Zeit abgebrochen wird. `belegen(Phase,DS)` wird deshalb als Subgoal von `doall/1` aufgerufen:

`doall(Goal)` ⟵ alle Erfüllungen von `Goal` wurden entwickelt. Dieses Prädikat ist immer erfolgreich.

`belegen/2` erzeugt vor allem Fakten `loesung/3`, die – wie gesagt – (Teil-) Lösungen der bereits berechneten Phasen speichern. Das ist auch schon die ganze Information, die wir von einer Belegung benötigen. Deshalb können wir die Belegung der Variablen anschließend rückgängig machen, was durch das Scheitern von `phase/2` geschieht, siehe letzte Zeile der Definition. `phase/2` scheitert also immer! Das ist der Grund für die Negation des rekursiven Aufrufs oben.

Wie `belegen/2` funktioniert, sehen wir in Abschnitt 5.3.5.

5.3.4 Constraints

Dieser Abschnitt beschreibt die Behandlung der Rahmenbedingungen des Problems und deren Darstellung durch IF/Prolog-Constraints. Zunächst folgt eine Darstellung des Prädikats, das die Constraints einträgt: `constraints_eintragen/3`. Dann werden in eigenen Abschnitten die Prädikate beschrieben, die `constraints_eintragen/3` benutzt.

`constraints_eintragen(Phase,Plan,DS)` ⟵ in die Variablen der Struktur `DS` sind bezüglich des bisherigen Plans `Plan` für die Phase `Phase` die Constraints eingetragen.

```
constraints_eintragen(Phase,Plan,DS) :-
        dsS0(DS,S0),
        S0 in [0,1],
```

```
dsS31(DS,S31),
s plan_eintragen(Plan,S31),

dsS32(DS,S32),
% zur Ermittlung der Belegungsgrenzen
transpose(Plan,PlanT),
s belegung_eintragen(Phase,PlanT,S32),

s einschraenkungen_eintragen(Phase,Plan,S31),

s wuensche_eintragen(Phase,10000,DS),

dsAT(DS,AusgleichTage),
s bereich_eintragen(Phase,AusgleichTage,S31).
```

`dsName(DS,Struktur)` ⟵ `Struktur` enthält die Struktur `Name` der Datenstruktur `DS`. Dabei ist `Name` der Name einer solchen Struktur, u.a. also `S0`, `S1`, `S31`, `S32` usw. Also gibt es entsprechend die Prädikate `dsS0/2`, `dsS1/2`, `dsS31/2`, `dsS32/2` usw.

Die folgenden Abschnitte beschreiben, was in `constraints_eintragen/3` im Einzelnen passiert.

Festlegen der Wertebereiche

Das Festlegen der Wertebereiche für die Domainvariablen ist einfach. Mit dem Operator `in/2` wird entweder für eine Variable oder für eine Liste von Variablen der Wertebereich definiert. Hier benutzen wir die Struktur S0 der Datenstruktur DS. S0 ist ja die Liste aller Domainvariablen. Mittels `S0 in [0,1]` wird damit für alle Variablen der boolesche Wertebereich $\{0,1\}$ festgelegt.

Bisherigen Plan eintragen

Der bisherige Plan, das ist die Teillösung aus der vorigen Phase, wird mit dem Prädikat `plan_eintragen/2` berücksichtigt.

`plan_eintragen(Plan,S31)` ⟵ in die Struktur `S31` ist der bisherige Plan `Plan`[2] eingetragen, das heißt, an allen Stellen, die nicht unbestimmt sind, an denen also der bisherige Plan nicht den Eintrag '`.`' hat, wird die entsprechende Variable mit einer Null belegt. `Plan` ist genau wie `S31` eine Liste von Zeilen.

[2] `S31` enthält wie `Plan` die Einträge in Zeilenlisten, damit stehen entsprechende Einträge an entsprechenden Stellen.

```
plan_eintragen(Plan,S31) :-
        mapper([Plan,S31],[PlanZ,S31Z],
            mapper([PlanZ,S31Z],[Bel,S],
                (Bel \= '.' -> S = 0 ;true))).
```

Hier sieht man auch schön die Verwendung von `mapper/3`: `Plan` und `S31` sind beides Listen von Listen und gleichmächtig. Das heißt, sowohl `Plan` als auch `S31` haben als Elemente für jede Zeile des Dienstplans eine Liste. Alle diese Listen sind gleich lang, nämlich mit einem Element für jede Spalte des Dienstplans. Der erste Aufruf von `mapper/3` ruft für entsprechende Elemente (an derselben Stelle in der Liste) `PlanZ` und `S31Z` (also die Zeilen) der Listen `Plan` bzw. `S31` nochmals `mapper/3` auf. Das wiederum ruft für entsprechende Elemente `Bel` und `S` der Listen `PlanZ` bzw. `S31Z` `Bel \= '.' -> S = 0 ;true` auf. Und damit wird die Variable `S` mit 0 belegt, falls die Belegung `Bel` der entsprechenden Zelle nicht '.', also nicht unbestimmt ist.

Belegungsgrenzen pro Tag

Wie wir in Abschnitt 5.2 gesehen haben, werden die Belegungsgrenzen pro Tag, das sind die Grenzen der Zuteilung von Schwestern zu Schichten des Tages, durch die Prädikate `belegung/4` spezifiziert.

Dabei muss allerdings zwischen zwei Varianten unterschieden werden: zwischen den Fakten für die erste Phase und denen für die anderen Phasen. Die Fakten für die erste Phase haben folgendes Format: `belegung(1,Tag,Min,Max)`. Für jeden Tag des Planungszeitraums gibt es einen solchen Fakt. Er definiert, wieviele Schwestern der Station an diesem Tag, d.h. in allen Schichten zusammen mindestens und höchstens zur Verfügung stehen sollen und damit von INTERDIP in der ersten Phase eine Arbeitsschicht, d.h. keine Freischicht, zugeteilt bekommen.

Die Fakten für die Belegungsgrenzen pro Tag für alle anderen Phasen haben die Form: `belegung(Phase,Rest,Min,Max)`, wobei `Phase > 1`. Hier steht `Rest` für die Anzahl der Schwestern, die in dieser Phase maximal noch verteilt werden können. Das heißt: werden in der ersten Phase an einem Tag `A` Schwestern zur Arbeit eingeteilt, erhalten also keine Freischicht, so gilt für die zweite Phase der Fakt `belegung(2,A,Min,Max)` und für diesen Tag gelten als Ober- und Untergrenze für die Belegung von Nachtschichten `Min` bzw. `Max`. Seien dann in der zweiten Phase `B` Schwestern in die Nachtschicht eingeteilt, stehen für die dritte Phase, also für Früh- und Spätschicht zusammen, noch `C is A-B` Schwestern zur Verfügung und dafür gilt dann der Fakt `belegung(3,C,Min,Max)`.

Insgesamt dient diese Darstellung der genauen Spezifikation der Ober- und Untergrenzen für die Belegung jeder Schicht, in Abhängigkeit der verfügbaren Schwestern.

Zur Ermittlung des „Rests“ werden in einer Phase für jede Spalte des Dienstplans die noch unbestimmten Zellen gezählt, die also, bei denen der (bisherige) Plan den Eintrag '.' hat. `Plan` war ja der beste Plan vor der Berechnung der aktuellen Phase und `PlanT` ist der transponierte Plan, also eine Liste von Spalten von Dienstplaneinträgen, genauso wie `S32` die Liste von Spalten der Domainvariablen ist. Unter- und Obergrenze der Belegung werden dann als Unter- und Obergrenze für die Summe über alle Domainvariablen einer Spalte verwendet.

`belegung_eintragen(Phase,PlanT,S32)` ⟵ Die Belegungsgrenzen pro Tag sind für Phase `Phase` in die Variablen `S32` bezüglich des bisherigen Plans `PlanT` eingetragen.

```
belegung_eintragen(Phase,PlanT,S32) :-
        mapper1([S32,PlanT],[S,PT],I,(
            const_sum(S,SS),
            % fuer die erste Phase hat belegung/4 ja
            % eine andere Bedeutung...
            ( Phase == 1 ->
                belegung(Phase,I,Min,Max)
            ;
                count(member('.',PT),SM),
                (I,'Belegungsverteilung') c
                    belegung(Phase,SM,Min,Max)
            ),
            (belegung,Phase,I) c SS ?>= Min,
            (belegung,Phase,I) c SS ?=< Max
        )).
```

Hier wird das Constraint `const_sum/2` und der Operator `c/2` benutzt:

`const_sum(Liste,Summe)` ⟵ `Summe` ist die Summe aller Elemente der Liste `Liste`, als Constraint verstanden!

`c(Id,Goal)` ⟵ `Goal` gilt. Gilt `Goal` nicht, und `Goal` ist in der Regel die Erklärung eines Constraints, wird dem Benutzer eine Meldung angezeigt, welcher Constraint wo (`Id`) nicht eingetragen werden konnte. Damit dient `c/2` der Anzeige und für den Benutzer der Lokalisierung von Inkonsistenzen des CSP (siehe). `c/2` ist als rechtsassoziativer Infix-Operator deklariert und bindet gerade stärker als der Komma-Operator und gerade schwächer als der `not`-Operator (genau wie `s/2`).

```
c(N,C) :- C,! ; inkonsistent(N,C),!,fail .
```

Globale Einschränkungen

Nach den Belegungsgrenzen werden die globalen Einschränkungen eingetragen.

`einschraenkungen_eintragen(Phase,Plan,S31)` ⟵ die globalen Einschränkungen für Phase `Phase` sind mittels Plan `Plan` in die Struktur `S31` eingetragen.

Die Formulierung des Prädikats `einschraenkungen_eintragen/3` ist zu aufwendig, um hier detailliert betrachtet werden zu können. Siehe dazu den Programmtext im Anhang (Datei `phasen.pro`).

Als Rahmenbedingung (siehe Abschnitt 5.2, `einschraenkung/2`) wird nur angegeben, welche der vorhandenen globalen Einschränkungen zur Planung verwendet werden sollen (die „eingeschalteten"). Die Einschränkungen selbst werden durch Fakten `einschraenkung_def/5` definiert:

`einschraenkung_def(Phase,Name,MusterPlan,MusterVar,Constraint)` ⟵ Die Einschränkung mit (dem eindeutigen) Namen `Name`, die in der Phase `Phase` verwendet wird, definiert für ein passendes Muster `MusterPlan` im Plan den Conatraint `Constraint` für entsprechende Variablen `MusterVar`.

Dazu zwei Beispiele:

- ```
  einschraenkung_def(2,
          'Nach zwei Nachtschichten mindestens zwei Freischichten',
          ['.','.',0,'.'|_],
          [A|_],
          A ?= 0).
  ```

  Diese Einschränkung wird in der zweiten Phase wirksam. Passt dort `MusterPlan` auf den bestehenden Plan (der ersten Phase), das heißt, gibt es im bestehenden Plan an aufeinanderfolgenden Tagen die Einträge `'.','.',0,'.'`, muss also eine Schwester an vier aufeinanderfolgenden Tagen zunächst zwei arbeiten, hat dann einen Tag frei und muss dann wieder arbeiten, so tritt diese Einschränkung in Kraft: in Phase zwei darf dann am ersten dieser vier Tage keine Nachtschicht zugewiesen und damit in die entsprechende Variable keine 1 eingetragen werden. Diese wird deshalb hier auf 0 gesetzt.

- ```
  einschraenkung_def(3,
          '11-Stunden-Regel (Spätschicht)',
          ['.','.'|_],
          [A,B|_],
          A ?>= B).
  ```

Diese Einschränkung definiert die 11-Stunden-Regel für die Spätschichten. In der dritten Phase werden ja alle noch unbestimmten Zellen auf Früh- und Spätschicht verteilt. Die Belegung einer Variable mit 0 bedeutet für den Dienstplan „Spätschicht“, mit 1 „Frühschicht“. Sind in der dritten Phase die Belegungen zweier aufeinanderfolgender Tage noch unbestimmt, also weder Freischicht noch Nachtschicht, muss der Wert der Variable für die erste Zelle größer oder gleich dem der Variable für die zweite Zelle sein. Das schließt eine Belegung der aufeinanderfolgenden Tage mit zuerst 0 und dann 1 bzw. zuerst Spätschicht und dann Frühschicht aus, was genau der 11-Stunden-Regel bezüglich Spätschicht entspricht.

Ähnlich den beiden Beispielen werden auf diese Art Einschränkungen definiert, wie „Nie mehr als 10 Tage am Stück arbeiten“ und die 11-Stunden-Regel für die Nachtschichten. Genauso können ähnlich geartete Einschränkungen definiert werden.

Die Einschränkungen werden also durch eine Art Muster (zu unterscheiden von den Mustern für die Belegung der Variablen, siehe Abschnitt 5.3.5) definiert. `einschraenkungen_eintragen/3` geht immer den bisherigen Plan durch und sucht passende (und per `einschraenkung/2` aktivierte) Einschränkungen, unifiziert dann an den entsprechenden Stellen die Variablen aus `MusterVar` mit denen aus `S31` und formuliert den zur Einschränkung angegebenen `Constraint`.

Wünsche

Die Wünsche werden mit dem Prädikat `wuensche_eintragen/3` eingetragen. Wünsche sind immer von der Art „Schwester N will am Tag T in der Schicht S arbeiten“. Ein Wunsch bezieht sich also immer auf genau eine Zelle des Dienstplans und läßt sich für jede Phase unmittelbar in eine Belegung der entsprechenden Variable übersetzen.

`wuensche_eintragen(Phase,Schwelle,DS)` $\longleftarrow$ alle Wünsche sind in die Phase `Phase` eingetragen und die Kostenschwelle für harte Wünsche ist `Schwelle`. Dieses Prädikat wird hier nicht in seiner Implementierung angegeben, weil das recht komplex ist und deshalb dessen Darstellung keinen Vorteil für das Verständnis bringt.

`wunsch(Phase,Wunsch,VarBel,AusgleichTage)` $\longleftarrow$ ist in der Phase `Phase` der Wunsch `Wunsch` geäußert, so bedeutet das eine Belegung der entsprechenden Variable mit `VarBel` und einen Ausgleich der Arbeitszeit im Planungszeitraum von `AusgleichTage` (siehe unten) für die entsprechende Schwester. Konkret sieht die Übersetzungstabelle wie folgt aus:

```
% Erste Phase: Frei/Dienst-Schichten
% Belegt werden "Dienstschichten", d.h. eine 1 steht
% fuer "Dienst", 0 fuer "Frei"
wunsch(1,'', _,0).
wunsch(1,'0',0,0).
wunsch(1,'1',1,0).
wunsch(1,'2',1,0).
wunsch(1,'3',1,0).
wunsch(1,'u',0,1).
wunsch(1,'d',1,0).

% Zweite Phase: Nachtschichten
wunsch(2,'', _,0).
wunsch(2,'0',0,0).
wunsch(2,'1',0,0).
wunsch(2,'2',0,0).
wunsch(2,'3',1,0).
wunsch(2,'u',0,0).
wunsch(2,'d',0,0).

% Dritte Phase: Fruehschichten
wunsch(3,'', _,0).
wunsch(3,'0',0,0).
wunsch(3,'1',1,0).
wunsch(3,'2',0,0).
wunsch(3,'3',0,0).
wunsch(3,'u',0,0).
wunsch(3,'d',_,0).
```

Wünsche werden ja mit den Fakten `wuensche/3` spezifiziert. Dabei wird für jede Kategorie von Wünschen, also für die *roten*, die *schwarzen* und die *weißen* Wünsche ein Wert für die Kosten angegeben, die entstehen (siehe Abschnitt 5.3.5), wenn ein Wunsch verletzt wird. Beim Eintragen der Wünsche werden alle Kategorien, deren Kosten gößer als die `Schwelle` sind, als *harte Wünsche* und die anderen als *weiche Wünsche* eingetragen. Die Schwelle ist, wie oben angegeben, 10000.

Harte Wünsche Harte Wünsche sind Wünsche, die unbedingt zu erfüllen sind. Welche das sind, legt der Benutzer fest. Er bestimmt, in welche Kategorie er konkrete Wünsche der Schwestern einträgt, und für welche Kategorie welche Kosten gelten sollen. Harte Wünsche werden entsprechend der Übersetzungstabelle `wunsch/4` in Variablenbelegungen umgewandelt, die beim Eintragen der Constraints unmittelbar angewendet werden.

Urlaub, und dazu zählt z.B. auch Mutterschaftsurlaub, ist immer als harter Wunsch einzutragen (siehe auch Abschnitt 2.1.1). Urlaub nämlich wird für die Schwester nicht als freier Tag gezählt, sondern als Arbeitstag. Hierfür werden die `AusgleichTage` aus der Übersetzungstabelle für jede Schwester summiert und beim Eintragen der Belegungsgrenzen pro Schwester berücksichtigt.

Würden Urlaubstage erst während der Belegung der Variablen und nicht schon während dem Eintragen der Constraints ermittelt, müsste auch die monatliche Arbeitszeit der Schwestern während der Belegung des Plans ständig überprüft werden. Wenn, wie hier, Urlaub aber schon vorher feststeht, können die minimale und maximale Anzahl von Schichten, die eine Schwester im Planungszeitraum tatsächlich zu arbeiten hat, schon vorher berechnet werden. Damit stehen die Belegungsgrenzen pro Schwester für die erste Phase (siehe unten) von vornherein fest und können als harte Constraints eingetragen werden, was die Propagation verbessert und damit den Suchraum weiter einschränkt.

Weiche Wünsche Weiche Wünsche können mangels der Mögichkeiten des IF/Prolog-Constraintsolvers, weiche Constraints zu behandeln, nicht als Constraints formuliert werden. Weiche Wünsche werden statt dessen zur Optimierung bestehender Lösungen verwendet. Das Verletzen eines weichen Wunsches erzeugt Kosten. Die weichen Wünsche werden genau wie die harten in Variablenbelegungen übersetzt. Diese werden aber nicht in Constraints umgewandelt, sondern werden gesammelt. Jede Zelle des Dienstplans (repräsentiert in Strukturen von DS) erhält eine Liste von Elementen der Form `Variablenbelegung/Kosten`. Die Kosten entstehen bei der Belegung des Dienstplans, wenn die tatsächliche Belegung einer Variablen nicht mit der Wunschbelegung an dieser Stelle übereinstimmt.

Belegungsgrenzen pro Schwester

Die Belegungsgrenzen pro Schwester werden ganz ähnlich den Belegungsgrenzen pro Tag behandelt. Die Grenzen selbst werden durch Fakten `bereich/4` definiert. Hier fließen noch die „Ausgleichstage“ mit ein.

`bereich_eintragen(Phase,AusgleichTage,S31)` ⟵ in die Variablen `S31` sind die Belegungsgrenzen pro Tag für die Phase `Phase` eingetragen und dabei `AusgleichTage` Ausgleichstage festgestellt worden.

```
bereich_eintragen(Phase,AusgleichTage,S31) :-
        mapper1([S31,AusgleichTage],[S,AT],I,(
            bereich(Phase,I,Min,Max),
            list_sum(S,SS),
            (bereich,Phase,I) c SS+AT ?>= Min,
```

```
            (bereich,Phase,I) c SS+AT ?=< Max
        )).
```

Hier wird die Summe über jede Zeile und damit die Anzahl der Schichten für jede Schwester nach unten und oben begrenzt. `AusgleichTage` ist die Anzahl der Urlaubstage, an denen die Schwester zwar frei hat, die aber als Arbeitstag gelten. Die Anzahl der tatsächlichen Arbeitstage plus die der Ausgleichstage ergibt damit die Anzahl der Arbeitstage der Schwester im Planungszeitraum. Und diese Summe wird nach oben und unten beschränkt.

Inkonsistenzen

Die letzten Abschnitte haben gezeigt, wie die Rahmenbedingungen des Dienstplanproblems technisch behandelt werden. Wir haben einige Stellen gesehen, an denen mittels `c/2` überprüft wird, ob ein Constraint oder eine sonstige harte Bedingung eingetragen werden kann. Gelingt das nicht, sind die Rahmenbedingungen inkonsistent. Werden zum Beispiel für eine Schwester sehr viele Freischichten als harte Wünsche eingetragen, kann evtl. die Mindestarbeitszeit der Schwester im Monat nicht mehr eingehalten werden, so dass die entsprechende Eintragung bei `bereich_eintragen/3` fehlschlägt. In diesem Fall ruft `c/2` das Prädikat `inkonsistent/2` auf:

`inkonsistent(Id,Goal)` $\longleftarrow$ die (eindeutige) Inkonsistenz `(Id,Goal)` (`Goal` schlug an der Stelle `Id` fehl) wird dem Benutzer mit erklärendem Text angezeigt. Dazu ein paar Beispiele, die sich eigentlich selbst erklären sollten (siehe auch `fehler/1`, Abschnitt 5.4.5 und `c/2`, Abschnitt 5.3.4):

```
inkonsistent((Schwester,Tag),wunsch(Phase,Wunsch,_,_)) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
            'Belegungsgrenzen pro Tag eingetragen.','\\n',
            'Globale Einschränkungen eingetragen.','\\n',
            '\\nINKONSISTENZ:\\n\\n',
            'Am Tag ',Tag,' kann für die Schwester ',
            Schwester,' der Wunsch ''',Wunsch,
            ''' nicht eingetragen werden.']).


inkonsistent((bereich,Phase,Schwester),(_ ?>= Min)) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
```

```
            'Belegungsgrenzen pro Tag eingetragen.','\\n',
            'Globale Einschränkungen eingetragen.','\\n',
            'Wünsche eingetragen.','\\n',
            '\\nINKONSISTENZ:\\n\\n',
            'Für Schwester ',Schwester,
            ' kann die minimale Belegung (',Min,
            ') nicht eingetragen werden.']).

inkonsistent((bereich,Phase,Schwester),(_ ?=< Max)) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
            'Belegungsgrenzen pro Tag eingetragen.','\\n',
            'Globale Einschränkungen eingetragen.','\\n',
            'Wünsche eingetragen.','\\n',
            '\\nINKONSISTENZ:\\n\\n',
            'Für Schwester ',Schwester,
            ' kann die maximale Belegung (',Max,
            ') nicht eingetragen werden.']).
```

5.3.5 Belegung der Variablen / des Dienstplans

Dieser Abschnitt beschreibt die Belegung der Variablen und damit die Erzeugung von Teillösungen für den Dienstplan.

`belegen(Phase,DS)` ⟵ für Phase `Phase` sind mit den Variablen aus `DS` optimale Lösungen erzeugt, soweit dies innerhalb der Zeitschranke `Z`, definiert durch `zeitschranke(Z)`, möglich war.

```
belegen(Phase,DS) :-
        dsSW31(DS,SW31),
        dsK(DS,Kosten),

        VorPhase is Phase-1,
        beste_loesung(VorPhase,Plan,_),

        zeitschranke(ZeitS),
        timebounded(
            minimize_maximum((
                belegen_plan(Phase,1,SW31,Kosten,Plan,Plan_),
                asserta(loesung(Phase,Plan_,Kosten)),
                ),Kosten),
            ZeitS)
        ).
```

`timbounded(Goal,Zeitschranke)` ⟵ `Goal` ist erfüllt oder nach `Zeitschranke` Sekunden unerfüllt abgebrochen.

`minimize_maximum(Goal,Werte)` ⟵ Goal ist erfüllt und das Maximum der Liste `Werte` ist minimiert. Das passiert mit einem Branch-and-Bound-Verfahren [Van89]: `Goal` wird entwickelt und anschließend müssen alle Elemente von `Werte` ganze Zahlen sein. Davon wird das Maximum festgestellt, `Goal` wird rückgängig gemacht, von allen Elementen von `Werte` wird das Maximum festgestellt und Goal wird nochmal entwickelt mit der Bedingung, dass alle `Werte` kleiner als das letzte Maximum sein müssen. Bei Erfüllung von `Goal` wird dieser Prozess wiederholt usw. `minimize_maximum/2` ist erfüllt, wenn keine Minimierung des Maximums der `Werte` mehr möglich ist.

`minimize_maximum/2` ist ein von IF/Prolog definiertes Prädikat [Sie96a].

`belegen_plan(Phase,Schwester,SW31,Kosten,Plan,Plan_)` ⟵ für Phase `Phase` und Schwester `Schwester` und alle folgenden Schwestern sind die Variablen `SW31` belegt und dadurch ist aus dem `Plan` der vorherigen Phase der Plan `Plan_` entwickelt, der mit `Plan` an allen Stellen bis auf die, die in `Plan` noch unbestimmt waren, identisch ist. Dort sind entsprechend der Belegung der Variablen aus `SW31` konkrete Einträge vorgenommen. Die dafür nötige Abbildung wird durch das Prädikat `phase_plan_map/3` definiert.

`phase_plan_map(Phase,Plan,Var,Plan_)` ⟵ in der Phase `Phase` wird aus dem Eintrag `Plan` des Plans der vorherigen Phase, wenn die entsprechende Variable mit dem Wert `Var` belegt wurde, der Eintrag `Plan_` im neuen Plan. Konkret sieht diese Abbildung wie folgt aus:

```
% phase_plan_map(Phase,Plan,Var,Plan_) :-
phase_plan_map(1,_,0,0).
phase_plan_map(1,_,1,_).
phase_plan_map(2,_,0,_).
phase_plan_map(2,_,1,3).
phase_plan_map(2,0,0,0).
phase_plan_map(3,_,0,2).
phase_plan_map(3,_,1,1).
phase_plan_map(3,0,0,0).
phase_plan_map(3,3,0,3).
```

Die Belegung der Variablen beruht auf der Struktur DS, die ja alle Variablen enthält und mit diesen alle jetzt für diese Phase formulierten Constraints. Für die Belegung wird die Darstellung der Variablen in Zeilenform zusammen mit den weichen Wünschen, `SW31`, und die Liste der Kosten,

`Kosten`, benötigt. Außerdem wird aus der Faktenbasis die beste Lösung der der Phase `Phase` vorausgehenden Phase `VorPhase`, `VorPhase is Phase-1`, geholt und davon der `Plan` (eine Teillösung als Liste von Zeilen mit Dienstplaneinträgen). Daraus werden mittels des Prädikats `timebounded/2` zeitbeschränkt und mittels des Prädikats `minimize_maximum/2` optimiert mittels des Prädikats `belegen_plan/6` neue Teillösungen erzeugt, `Plan_`, und diese werden in die Faktenbasis eingetragen.

`belegen_plan/6` belegt eine Zeile (die der Schwester `Schwester`) und ruft sich dann selbst rekursiv für die nächste Schwester auf. Damit wird der Dienstplan zeilenweise belegt. Bei der Belegung sind zwei Dinge interessant: einmal die Verwendung der Muster. Befindet sich die Belegung an einer Stelle des Plans, wird für jedes Muster dieser Phase, in der Reihenfolge, in der die Muster in der Faktenbasis stehen, versucht, zuerst die Kosten für das Muster (Abschnitt 5.2) zu den bisherigen Kosten für diese Zeile (Kosten entstehen ja zeilenweise) hinzuzuaddieren. Gelingt das, und das kann durch Überschreitung der Kostengrenze (`minimize_maximum/2`) fehlschlagen, wird versucht, das Muster vollständig in den Plan einzutragen. Gelingt das für ein Muster, und hier ist es wichtig, dass für jede Phase auch triviale Muster existieren, die immer eingetragen werden können, wird an der Stelle nach dem letzten Eintrag des Musters fortgefahren. Ist eine Zeile innerhalb der Belegung eines Musters zu Ende, gilt die bisherige Belegung des Anfangs des Musters als erfolgreich, das Muster wird also verwendet, aber nicht vollständig.

`kosten_wuensche(Belegung,WunschListe,Kosten)` ⟵ `Kosten` sind die Kosten, die sich durch die instantiierte Variable `Belegung` und die weichen Wünsche `WunschListe` ergeben.

```
% kosten_wuensche(Belegung,WunschListe,Kosten) :-
kosten_wuensche(B,[WK|R],Kosten) :-
        not var(WK),
        !,
        WK=W/K,
        ( B=W ->        % der Trigger !
                Kosten = Kosten_
        ;
                Kosten ?= K + Kosten_ ),
        kosten_wuensche(B,R,Kosten_).
kosten_wuensche(_,_,K) :- !,K=0.
```

Der andere interessante Punkt ist hier die Behandlung der Kosten für weiche Wünsche: mit dem Prädikat `kosten_wuensche/3`. Hier wird an jeder Stelle des Dienstplans die aktuelle Variablenbelegung nacheinander mit allen in die Wunschliste (Abschnitt 5.3.4)eingetragenen verglichen, und wenn die nicht übereinstimmen, genauer: wenn die aktuelle Variable mit dem Eintrag in der Wunschliste nicht unifiziert werden kann, werden die zugehörigen Kosten für die Verletzung des Wunsches zu den Kosten dieser Zeile hinzuaddiert. Hier besteht eine einfache Möglichkeit der Erweiterung von INTERDIP: wäre `W` in `kosten_wuensche/3` nicht ein einfacher Wert für die Belegung der Variable, sondern die Variable einer anderen Zelle des Dienstplans, dann wäre das die Implementierung eines Wunsches wie „Schwester Anna möchte (nicht) mit Schwester Berta in einer Schicht arbeiten“.

5.4 Benutzerschnittstelle

Dieser Abschnitt beschreibt die Realisierung der Benutzerschnittstelle von INTERDIP. Diese wird nicht so detailliert wie die Prolog-Implementierung des eigentlichen Dienstplaners beschrieben. Vielmehr wird auf die spezifischen Probleme und deren Lösung der Implementierung einer interaktiven Benutzerschnittstelle mit Prolog eingegangen. Der grafische Teil der Schnittstelle wurde in Tcl/Tk realisiert. SNI hat speziell für diese Arbeit eine Tcl/Tk-Schnittstelle zu ihrem IF/Prolog realisiert.

Die folgenden Unterabschnitte beschreiben zunächst kurz Tcl/Tk, dann die Tcl/Tk-IF/Prolog-Kombination `tkprolog`, und nach einem kurzen Hinweis auf die Programmierung der Eingabefenster von INTERDIP den Debugger und die händische Bearbeitung und Konsistenzprüfung.

5.4.1 Tcl/Tk

Tcl und Tk sind zwei Software-Pakete, die John K. Ousterhout, der Vater von Tcl/Tk, wie folgt beschreibt ([Ous94]):

> Zusammen bilden sie ein Programmiersystem zur Entwicklung und Benutzung von grafischen Benutzeroberflächen. Tcl ist eine einfache Skriptsprache (*scripting language*) zur Kontrolle und Erweiterung von Anwendungen, sein Name steht für *tool command language*. Tcl bietet generische Programmiermöglichkeiten wie Variablen, Schleifen und Prozeduren, die nützlich sind für eine Vielzahl von Anwendungen. Außerdem bietet Tcl die Möglichkeit der Einbettung (*embedding*).Sein Interpreter ist eine Bibliothek von C-Prozeduren, die leicht in Anwendungen integriert werden können und jede Anwendung kann den Kern von Tcl mit zusätzlichen Kommandos erweitern.
>
> Eine der nützlichsten Erweiterungen für Tcl ist Tk, ein Werkzeugkasten für das X-Window System. Tk erweitert den Kern von Tcl um Kommandos zur Erzeugung von Benutzerschnittstellen, so dass man Motif-ähnliche Schnittstellen bauen kann, indem man Tcl Skripten schreibt anstatt C-Code. Wie Tcl ist auch Tk

> als Bibliothek von C-Prozeduren realisiert, so dass es in vielen verschiedenen Anwendungen benutzt werden kann. Individuelle Anwendungen können auch die grundlegenden Tk-Eigenschaften erweitern um in C geschriebene grafische Elemente und Geometriemanager.

Tcl ist also eine Programmiersprache, gedacht als Skriptsprache, also zur Programmierung bestehender Softwarepakete. Das heißt, damit können bestehende Pakete leicht programmierbar gemacht werden. Das wichtigste Paket ist Tk. Mit Tcl/Tk lassen sich ganze Programme schreiben und vor allem grafische Benutzeroberflächen für bestehende Anwendungen programmieren. Tcl ist interpretiert und die Tk-Prozeduren sind sehr mächtig. Dadurch lassen sich mit wenigen Zeilen einfache bis mittelmäßig komplexe Benutzerschnittstellen realisieren, ohne dass ständig Quellcode übersetzt werden müsste. Und die Einbindung von Tcl/Tk in andere Systeme ist wirklich einfach. In der Regel genügen wenige C-Funktionen.

Insgesamt also eignet sich Tcl/Tk bestens zur Erweiterung bestehender Software um grafische Möglichkeiten und zur schnellen prototypischen Entwicklung grafischer Benutzerschnittstellen. Deshalb wurde für die Realisierung von INTERDIP von SNI eine Tcl/Tk-Schnittstelle für IF/Prolog implementiert.

5.4.2 tkprolog

`tkprolog` ist das kombinierte Tcl/Tk-IF/Prolog System. `tkprolog` enthält eine Prolog-Maschine und einen Tcl-Interpreter zusammen mit der Tk-Bibliothek. `tkprolog` enthält also zwei Interpreter. Beide sind insofern unabhängig voneinander, als sie verschiedene Daten- und Programmspeicher haben. Der Prolog-Interpreter kann also Prolog-Dateien laden und Prädikate speichern, und der Tcl-Interpreter genauso Dateien laden und Prozeduren speichern. Diese sind für den jeweils anderen unzugänglich. Es ist aber auch kein Konzept zur gemeinsamen Datenhaltung realisiert. Die Komunikation zwischen den Interpretern und damit zwischen Programmteilen der Anwendung, die in Tcl und solchen die in Prolog programmiert sind, passiert über eine handvoll Tcl-Prozeduren und Prolog-Prädikate. Zu einer Zeit ist immer nur ein Interpreter aktiv, also können nie zwei Anweisungen parallel ausgeführt werden. Nach dem Aufruf von `tkprolog` ist zunächst der Prolog-Interpreter aktiv, der die in der Kommandozeile angegebenen Dateien lädt und dann das Prädikat `program/0`, soweit vorhanden, aufruft.

Tcl/Tk ist ereignisgesteuert. Das heißt, der Tcl-Interpreter wartet auf Ereignisse von außen (in der Regel von der Tk- bzw. X-Window-Bibliothek) und reagiert dann darauf. Tcl/Tk befindet sich also immer in der Schleife *auf Ereignis warten – auf Ereignis reagieren*. Diese Schleife kann aber auch von IF/Prolog nachgebildet werden, was wir hier tun müssen. Sehen wir uns nochmal das „Hauptprogramm“ an:

```
program :-
            tk([file('interdip.tcl')]),

            s erz_ds(DS),

            repeat,
            tk_do_one_event(0),
            (current_global(phase) ->
                get_global(phase,Phase),
                phase s Phase s not phase(Phase,DS),
                unset_global(phase)
            ;
                unset_global(phase)
            ),
            (current_global(konsistenz_pruefen) ->
                get_global(konsistenz_pruefen,Plan),
                konsistenz_pruefen(Plan,DS),
                unset_global(konsistenz_pruefen)
            ;
                unset_global(konsistenz_pruefen)
            ),
            fail.
```

`tk(Optionen)` ⟵ `Optionen` ist eine Liste von Optionen, mit denen das Tk-Paket und mit diesem der Tcl-Interpreter initialisiert wird.

`tk_do_one_event(Maske)` ⟵ Tk wartet[3] auf *ein* durch `Maske` maskiertes Ereignis und reagiert dann darauf. `Maske == 0` steht für alle Ereignisse.

`tk_loop_while_events` ⟵ alle für Tk anstehende Ereignisse sind (damit) abgearbeitet.

Hier wird zuerst Tk initialisiert und dafür die Datei `interdip.tcl` geladen. `interdip.tcl` enthält Definitionen von Prozeduren, die den wesentlichen Teil der Benutzerschnittstelle von INTERDIP implementieren, und Befehle zum Laden der restlichen Definitionen.

Nach Erzeugung der Variablen (siehe Abschnitt 5.3) folgt eine unendliche Schleife: in ihr wird zunächst mittels `tk_do_one_event/1` auf ein Ereignis

[3] Das Prädikat `tk_do_one_event/1` ist *blockierend*, das heißt, dass, solange kein Ereignis vorliegt, das ganze Programm INTERDIP angehalten ist. Das ist aber hier auch sinnvoll: erst wenn z.B. der Benutzer ein Ereignis auslöst, indem er z.B. einen Menübefehl anklickt, muss das Programm wieder etwas tun. Bis dahin verbraucht es nicht unnötig Rechenkapazität.

gewartet, das dann ausgeführt wird. Hat der Benutzer eine Berechnung angestossen, wird eine entsprechende globale Variable gesetzt, die dann zum Aufruf von vor allem `phase/2` führt. Diese etwas ungewöhnliche Behandlung eines Aufrufs, ist notwendig durch die in Abschnitt 5.3.2 geschilderte Tatsache, dass Domainvariablen nicht in der Faktenbasis abgelegt werden können, weil sie dadurch alle Constraints und Wertebereiche verlieren. Dadurch muss die Struktur DS ab der Erzeugung an alle Prädikate als Parameter mitgegeben werden, die sie benötigen. Und genau das wird durch diese Implementierung ermöglicht: `program/0` erzeugt mittels `erz_ds/1` die Struktur DS und kann diese dann an die Prädikate, die DS benötigen, weitergeben.

Neben `phase/2` (Abschnitt 5.3.3) wird in der Schleife `konsistenz_pruefen/2` aufgerufen: siehe Abschnitt 5.4.5.

5.4.3 Eingabefenster

Die Benutzerschnittstelle von INTERDIP benutzt zur Anzeige eines Dienstplans, zur Eingabe von Rahmenbedingungen usw. verschiedene Fenster. Es ist in Tcl/Tk sehr einfach, Menüs zu generieren, Fenster mit Ein- und Ausgabebausteinen in vielen Formaten zu erstellen, und diese alle mit Funktion zu versehen. Entsprechend einfach ist der Aufbau der Eingabefenster von INTERDIP. Fast alles, was für die Programmierung der Fenster und damit für den größten Teil der Benutzerschnittstelle von INTERDIP, nötig war, ist in jeder Einführung in Tcl/Tk beschrieben: [Ous94, Wel95].

5.4.4 Debugger

Der Debugger besitzt folgende Fähigkeiten: Anhalten einer laufenden Berechnung, Anhalten an einem vorher definierten Breakpoint, Rücksprung zu einer vom Benutzer festzulegenden Zelle des Dienstplans und Auswahl einer konkreten Dienstplanbelegung durch den Benutzer. Technisch interessant sind hier die Möglichkeiten, eine Berechnung anzuhalten, und der Rücksprungmechanismus.

Während der Berechnung einer Phase befindet sich `tkprolog` im Prolog-Interpreter, auf Aktionen des Benutzers in der Benutzerschnittstelle wird damit nicht reagiert. Dafür wird während der Berechnung regelmäßig `tk_loop_while_events/0` aufgerufen, das anstehende Ereignisse verarbeitet. Konkret passiert das mit dem Prädikat `process_events`, das von `belegen_plan/2` (siehe Abschnitt 5.3.5) aufgerufen wird. Tatsächlich existieren zwei Versionen von `belegen_plan/2`: eine ohne und eine mit Debugger. Dafür existieren zwei Versionen von `process_events`: `process_events/0` und `process_events/1`.

`process_events` $\longleftarrow$ evtl. anstehende Tcl/Tk-Ereignisse sind verarbeitet und auf evtl. Abbruch der Berechnung ist reagiert.

```
process_events :-
        tk_loop_while_events,
        process_stop,
        !.
```

`process_events(DebugPoints)` ⟵ `DebugPoints` ist eine Liste möglicher Rücksprungpunkte in der Berechnung der Phase und evtl. anstehende Tcl/Tk-Ereignisse sind verarbeitet, falls dann die Pause eingeschaltet ist (`current_global(pause)`), ist diese mittels `process_events_pause/1` verarbeitet und auf evtl. Abbruch der Berechnung ist reagiert und evtl. Rücksprung ist mittels `zurueck/1` ausgelöst. Dieses Prädikat entspricht `process_events/0` und wird statt diesem benutzt, falls der Debugger aktiv ist.

```
process_events(DebugPoints) :-
        tk_loop_while_events,
        (current_global(pause) ->
            process_events_pause(DebugPoints) ; true),
        process_stop,
        zurueck(DebugPoints),
        !.
```

`process_events_pause(DebugPoints)` ⟵ `process_events/1` und Pause ist behandelt. Sollte die Pause, also die Unterbrechung der Berechnung, nicht eingeschaltet sein, wird das zunächst gemacht. Darauf folgt eine `repeat`-Schleife, die erst beendet ist, wenn die Pause ausgeschaltet ist (`not current_global(pause)`). Im Unterschied zu `process_events/0` und `process_events/1`, die ja während einer laufenden Berechnung ohne Pause ausgeführt werden, benutzt `process_events_pause/1` zur Abfrage und Ausführung der Tcl/Tk-Ereignisse das Prädikat `tk_do_one_event/1`, das die Ausführung solange blockiert, bis tatsächlich ein Ereignis vorliegt.

```
process_events_pause(DebugPoints) :-
        pause_ein,
        repeat,
        tk_do_one_event(0),
        process_stop,
        zurueck(DebugPoints),
        not current_global(pause),
        !.
```

`process_stop` ⟵ wenn der Schalter (die globale Variable) `process_events_stop` gesetzt ist, wird das Prolog-Ereignis `timeout` ausgelöst. Das führt, genau wie eine Zeitüberschreitung (siehe Abschnitt 5.3.5), zum Abbruch der Berechnung der aktuellen Phase.

```
process_stop :-
        current_global(process_events_stop),
        !,
        unset_global(process_events_stop),
        throw(timeout).
process_stop.
```

`zurueck(DebugPoints)` ⟵ falls der Schalter `zurueck`, eine globale Variable, gesetzt ist (und das ist der Fall, falls der Benutzer eine Stelle im Dienstplan oder den Befehl zum Rückgängigmachen eines Belegungsschritts angeklickt hat), dann hat die globale Variable `zurueck` den Wert `(Schwester,Tag)`, der die angeforderte Rücksprungstelle bezeichnet. `DebugPoints` enthält alle in der aktuellen Belegung möglichen Rücksprungpunkte (wird bei `belegen/2` ein Muster belegt, ist die erste Zelle der Belegung ein Rücksprungpunkt; das heißt, es kann nur an den Anfang eines belegten Musters zurückgesprungen werden). Der erste aller möglichen Rücksprungpunkte, der vor der angeforderten Stelle liegt, wird als Prolog-Ereignis ausgelöst. Damit wird zurückgesprungen.

```
zurueck(DebugPoints) :-
        current_global(zurueck),
        !,
        get_global(zurueck,(Schwester,Tag)),
        unset_global(zurueck),
        member(d(S,T),DebugPoints),
        T =< Tag,
        S =< Schwester,
        tcl('verfolgen ## ## white',[S,T]),
        tk_loop_while_events,
        throw(d(S,T)).
zurueck(_).
```

Für Rücksprünge benötigen wir die Möglichkeit, innerhalb Prolog Ereignisse zu definieren und auszulösen, wodurch an bestimmte Stellen im Programm gesprungen werden kann. Dafür definiert IF/Prolog ISO-gemäß vor allem die zwei Prädikate `context/2` und `throw/1`. Die dadurch definiert Ereignisse sind im übrigen Prolog-Ereignisse, beenflussen also die Ausführung des Prolog-Programms durch den Interpreter und sind von den Tcl/Tk-Ereignissen zu unterscheiden!

`context(Goal,Liste)` ⟵ `Goal` wird im Kontext `Liste` ausgeführt. Ist ein Element von `Liste` von der Form `catch(Term) => SubGoal`, dann wird, sollte innerhalb der Ausführung von `Goal` das Ereignis `Term` ausgelöst werden (`throw(Term)`), wird die Ausführung von Goal sofort rückgängig gemacht und durch die Ausführung von `SubGoal` ersetzt.

`throw(Term)` ⟵ das Ereignis `Term` wird ausgelöst. Dieses muss durch `context/2` definiert sein, sonst tritt ein Fehler auf.

5.4.5 Händische Bearbeitung und Konsistenzprüfung

Die händische Bearbeitung eines generierten Dienstplans benutzt das Hauptfenster der Anwendung zur Bearbeitung des angezeigten Plans durch den Benutzer. Zur Anzeige von Inkonsistenzen (siehe auch Abschnitt 5.3.4) dient das Prädikat `fehler/1`.

`fehler(Text)` ⟵ `Text` ist eine Liste von Wörtern, die dem Benutzer als Fehlermeldung angezeigt wird.

```
fehler(L) :-
        tcl_string(L,Ls),
        tcl('fehler ##',[Ls]).
```

`tcl_string(Term,String)` ⟵ `Term` ist ein Prolog-Term und `String` der entsprechende Tcl-String (als Prolog-Atom). Dabei wird z.B. eine Prolog-Liste in eine Tcl-Liste umgewandelt.

`tcl(String,Parameter)` ⟵ `String` ist ein Tcl-String (als Prolog-Atom) und `Parameter` ist eine Liste von Parametern, die in `String` für `'##'` ersetzt werden. Der daraus resultierende String wird vom Tcl-Interpreter ausgewertet.

Die eigentliche Konsistenzprüfung eines bearbeiteten Dienstplans geschieht mit dem Prädikat `konsistenz_pruefen/2`:

`konsistenz_pruefen(Plan,DS)` ⟵ der Plan `Plan` ist mithilfe der Datenstruktur `DS` auf Konsistenz überprüft und eventuelle Inkonsistenzen sind dem Benutzer angezeigt.

`konsistenz_pruefen/2` stützt sich genau wie `phase/2` auf die Prädikate `constraints_eintragen/3` und `belegen/2`. Zum einfacheren Verständnis davon, wie `konsistenz_pruefen/2` funktioniert, seien die Erzeugung eines kompletten Dienstplans in allen drei Phasen und die Konsistenzprüfung als Prädikate dargestellt, die es so allerdings in INTERDIP nicht gibt:

```
plan_erzeugen(Plan3) :-
        var(Plan3),

        Plan0 = [.,.,.],

        variablen_erzeugen(Vars1),
        constraints(phase(1),Plan0,Vars1),
        belegen(phase(1),Plan0,Vars1,Plan1), % Plan0 -> Plan1

        variablen_erzeugen(Vars2),
        constraints(phase(2),Plan1,Vars2),
        belegen(phase(2),Plan1,Vars2,Plan2), % Plan1 -> Plan2

        variablen_erzeugen(Vars3),
        constraints(phase(3),Plan2,Vars3),
        belegen(phase(3),Plan2,Vars3,Plan3). % Plan2 -> Plan3
```

Die drei Variablensätze `Vars1` bis `Vars3` symbolisieren die Verwendung der Datenstruktur DS und das Rückgängigmachen erfolgreicher Belegungen der Variablen. Man kann die Planung also übersichtshalber darstellen als: ausgehend vom trivialen Plan der Phase 0, in dem alle Zellen unbestimmt sind, werden mit Variablen `Vars1` Cosntraints formuliert und anschließend unter Belegung der Variablen ein neure Plan `Plan` erzeugt. Daraus wird genauso `Plan2` und `Plan3`.

Die Konsistenzprüfung geht genau umgekehrt vor:

```
konsistent(Plan3) :-
        not var(Plan3),

        variablen_erzeugen(Vars3),
        belegen(phase(3),Plan2,Vars3,Plan3), % Plan3 -> Plan2
        constraints(phase(3),Plan2,Vars3),

        variablen_erzeugen(Vars2),
        belegen(phase(2),Plan1,Vars2,Plan2), % Plan2 -> Plan1
        constraints(phase(2),Plan1,Vars2),

        variablen_erzeugen(Vars1),
        belegen(phase(1),Plan0,Vars1,Plan1), % Plan1 -> Plan0
        constraints(phase(1),Plan0,Vars1),

        Plan0 = [.,.,.].
```

Ausgehend von einem vollständigen `Plan3` wird unter Benutzung desselben Prädikats zur Belegung eines Plans wie bei der Planung oben, ein `Plan2` erzeugt. Hier sieht man einen Vorteil der deklarativen Programmierung des `belegen`-Prädikats: es kann aus einem unvollständigeren Plan P unter Belegung von Variablen einen vollständigeren machen P' aber auch umgekehrt aus P' den unvollständigeren Plan P. Die zweite Richtung ist übrigens eindeutig: der Plan P' unterscheidet sich von P nur an Stellen, die in P unbestimmt sind, alle anderen Stellen (oder Zellen) sind in P und P' identisch. Umgekehrt heißt das, dass alle die Stellen die in einer Phase eine bestimmte Belegung bekommen, und die sind eindeutig bestimmt und nach der Phase an der Belegung selbst zu erkennen, vor der Phase unbestimmten Inhalt haben mussten.

Unter Belegung der Variablen wurde also aus `Plan3` der `Plan2` erzeugt. Und für die belegten Variablen könne jetzt die Constraints eingetragen werde. Wenn ein Constraint nicht eingetragen werden kann, ist die entsprechende Stelle des Dienstplans inkonsistent und die Art des Constraints gibt Auskunft über die Art der Inkonsistenz. Siehe hierzu auch Abschnitt 6.2.4.

5.5 Mögliche Erweiterungen

Ziel dieser Arbeit war ja auch ein Werkzeugkasten zum Entwurf anderer Problemlöser. Die verwendeten Konzepte und implementierten Prädikate sollten also möglichst allgemein verwendbar sein. Insbesondere sollte es deshalb nicht schwer sein, den bestehenden Dienstplaner um neue Funktionalität zu erweitern.

Dieser Abschnitt beschreibt kurz, wie mögliche Erweiterungen realisiert werden können.

5.5.1 Weitere Wunsch-Kategorien

Weitere Wunschkategorien (neben *rot*, *schwarz* und *weiß*) erfordern eigentlich nur eine Erweiterung der Benutzerschnittstelle um entsprechende Eingabefenster. Der Planer selbst bezieht sich auf eine beliebige Zahl von `wunsch/3`-Fakten, die ja auch die Kosten für die Verletzung der Wünsche beschreiben und damit, ob die Wünsche der Kategorie als hart oder weich zu betrachten sind.

5.5.2 Komplexe Wünsche

In Abschnitt 2.1.4 waren ja noch andere Wünsche angedeutet, als die in INTERDIP implementierten: komplexere Wünsche. Am Ende des Abschnitts 5.3.5 wurde die Möglichkeit der Erweiterung der weichen Wünsche bereits angedeutet. Die harten Wünsche können tatsächlich eher noch einfacher erweitert werden.

- „Wenn ich am n-ten Tag des Monats arbeiten muss, dann möchte ich am m-ten Tag frei haben“: in der ersten Phase muss die Summe aus dem Wert der Variablen des n-ten Tags und dem Wert der Variablen des m-ten Tags kleiner oder gleich 1 sein (es dürfen nicht beide 1 sein).

- „Schwester Anna möchte (nicht) mit Schwester Berta in einer Schicht arbeiten“: In jeder außer der ersten Phase muss für alle Tage die Summe der Werte der entsprechenden Variablen der beiden Schwestern kleiner oder gleich 1 sein (es dürfen nicht beide 1 sein).

Für komplexe Wünsche braucht es vor allem eine passende Möglichkeit der Beschreibung bzw. Spezifikation.

5.5.3 Weitere Schichten

Jede weitere Schicht erfordert eine weitere Phase. Dazu müssen die Prädikate der Spzifikation `bereich/4`, `belegung/4` und `muster/3` um entsprechende Einträge und die Eingabemöglichkeiten der Benutzerschnittstelle erweitert werden. Außerdem muss das Menü zum Anstoßen der Phasen um einen Eintrag und nicht zuletzt die Abbildung `phasen_plan_map/4` und evtl. die globale Einschränkungen `einschraenkung_def/5` erweitert werden. Technisch ist das alles trivial.

Allerdings ist die Wahl der Schicht und – damit zusammenhängend – der Muster und evtl. der Einschränkungen nicht ganz einfach. So wurde zum Beispiel die dritte Schicht, die Nachtschicht, nicht zuletzt deshalb in der zweiten Phase belegt, weil dadurch feststeht, dass alle folgenden Schichtbelegungen entweder eine Früh- oder eine Spätschicht ergeben und mit der 11-Stunden-Regel damit klar ist, dass nach einer Nachtschicht nur wieder eine Nachtschicht oder aber eine Freischicht, die nach der ersten Phase ja auch schon bekannt sind, kommen darf, also insgesamt in der zweiten Phase auf eine Nachtschicht keine unbestimmte Schicht folgen darf. Dadurch kann die 11-Stunden-Regel für die Nachtschichten mit dieser Modellierung relativ einfach als Constraint ausgedrückt werden.

Bei der Modellierung neuer Schichten muss unter anderem darauf Rücksicht genommen werden, und das ist nicht trivial.

Kapitel 6

Handbuch

Dieses Kapitel enthält das Handbuch zur Benutzung von INTERDIP. Das Handbuch ist ursprünglich eine html-Datei und ist eigentlich in der elektronischen Form zu benutzen.

Alles, was folgt, ist aus der html-Datei durch Übersetzung mit dem Paket `html2tex` (siehe URL `http://www.cs.utwente.nl/~faase`)entstanden. Die für die elektronische Form so wichtigen Links wurden in Referenzen auf die daraus erzeugten Abschnitte übersetzt. Der Text eines html-Links ist in dieser Papierversion unterstrichen, ihm folgt die Nummer des Abschnitts, auf den er verweist.

Handbuch INTERDIP

InterDip Dieses Handbuch gibt die Anleitung zur Benutzung von INTERDIP, dem interaktiven Dienstplaner: INTERDIP[1]

Hans Schlenker, 4.8.1997

6.1 Benutzung des Handbuchs

Das Handbuch ist auf zwei verschiedene Arten zu benutzen:

- anhand der Darstellung der Dienstplanung (6.2) mit INTERDIP, oder
- entlang der INTERDIP-Fenster, ausgehend vom Hauptfenster (6.3), deren Abbildungen zu den enstprechenden Kapiteln führen.

Das Handbuch besteht aus nur einer Seite. Dadurch kann es leicht durchsucht werden und spart sich einen umständlichen Index. Es ist stark verkettet: die Abbildungen sind "clickable images", d.h., wenn man in eine Abbildung klickt, kommt man im Handbuch automatisch zu der Stelle, an

[1]Siehe URL http://www.pms.informatik.uni-muenchen.de/.schlenkh/interdip.html

der der entsprechende Teil des Bildes erklärt ist. Zusätzlich gibt es im Text Verweise auf diese Stellen.

Sollte tatsächlich einmal eine Verbindung nur in einer Richtung existieren, also vom Ziel eines Links keine Verbindung zurück zur Definition existieren, kann mit dem in jedem Browser verfügbaren Befehl ¨Back¨ zurückgesprungen werden. Hilfreich kann hier auch die Liste der letzten Adressen sein (Netscape: Menü: Go).

6.2 Dienstplanung

Die Generierung eines Dienstplans sieht typischerweise wie folgt aus:

1. Laden einer Einstellungsdatei (6.47)
2. Eingabe der Rahmenbedingungen (6.2.2)
3. Berechnen (6.2.3) der 1. Phase (6.42)
4. Berechnen (6.2.3) der 2. Phase (6.43)
5. Berechnen (6.2.3) der 3. Phase (6.44)

Wird in einer der Phasen, also bei einer Berechnung, keine Lösung gefunden, müssen die Rahmenbedingungen (6.2.2) verändert werden. Was genau zu verändern ist, steht unter Konsistenz (6.2.4).

6.2.1 Beispiel einer konkreten Dienstplanung

Die Abbildungen des Handbuchs bilden zusammen ein konkretes Beispiel einer Dienstplanung. Unter den gegebenen Rahmenbedingungen (6.2.2) generierte INTERDIP auf einem durchschnittlichen Rechner mit einer Zeitschranke von 10 Sekunden die Phasen Eins (6.42), Zwei (6.43) und Drei (6.44) und damit einen vollstaendigen Dienstplan. Das kann mit der INTERDIP-Testversion[2] sofort nachvollzogen werden. Die Einstellungen stammen aus der Einstellungsdatei H1.10.14.s

Siehe auch die einzelnen Bemerkungen zu:
Rahmenbedingungen (6.2.2) ,
Berechnen einer Phase (6.2.3) ,
Konsistenz (6.2.4) ,
Veränderte Rahmenbedingungen und bestehende Teillösung (6.2.5) ,
Zur Bezeichnung der Wünsche (6.15.1) ,
Optimierung eines Dienstplans (6.16.1) ,
Behandlung der Wochenenden (6.17.1) ,
Kosten für die Wünsche (6.18.1) ,
Länge der Schichten (6.20.1) ,

[2]Siehe URL http://www.pms.informatik.uni-muenchen.de/.schlenkh/interdip.html

Globale Einschränkungen (6.28) ,
Benutzung der Muster (6.29.1) ,
Belegung des Dienstplans (6.29.2) ,
Kosten (6.45.1) und
Einstellungsdateien (6.47.1) .

6.2.2 Rahmenbedingungen

Die Rahmenbedingungen für die Dienstplanung bestehen aus:

- Wünschen in drei Kategorien: rot (6.15), schwarz (6.16) und weiß (6.17)
- Bereichen:
  - je Schwester für Phase 1 (6.20), Phase 2 (6.21) und Phase 3 (6.22)
  - je Tag für Phase 1 (6.23), Phase 2 (6.24) und Phase 3 (6.25)
  - globalen Einschränkungen (6.28)
- Mustern für Phase 1 (6.29), Phase 2 (6.30) und Phase 3 (6.31)

Achtung: siehe auch Veränderte Rahmenbedingungen und bestehende Teillösung (6.2.5)

6.2.3 Berechnen einer Phase

In den einzelnen Phasen werden Lösungen generiert und optimiert. Zu Beginn einer Phase werden die Rahmenbedingungen (6.2.2) für diese Phase auf Konsistenz (6.2.4) überprüft. Sind sie konsistent, wird eine Belegung für diese Phase gesucht und optimiert (6.16.1). Nach der Zeitschranke (6.14) wird die Berechnung automatisch beendet. In die Berechnung selbst kann der Benutzer mit dem Debugger (6.34) eingreifen. Danach kann der Benutzer den erzeugten Plan bearbeiten (6.36) und dann die nächste Phase anstoßen. Siehe Phase 1 (6.20), Phase 2 (6.21) und Phase 3. (6.22)

Achtung: siehe auch Veränderte Rahmenbedingungen und bestehende Teillösung (6.2.5)

6.2.4 Konsistenz

Konsistenz ist die Korrektheit aller Rahmenbedingungen (6.2.2) untereinander bzw. die Korrektheit eines Dienstplans bezüglich aller Rahmenbedingungen. Konsistenz wird vor dem Belegen eines Dienstplans (6.29.2) überprüft. Außerdem kann der Benutzer einen selbst bearbeiteten Dienstplan (6.36) auf Konsistenz überprüfen lassen. Sind die Rahmenbedingungen oder ist ein Dienstplan offensichtlich inkonsistent, zeigt INTERDIP das an und möglichst genau auch die Inkonsistenz selbst. Welche das sind, und wie der Benutzer eine solche beseitigen kann, siehe

- Inkonsistenz: Belegung pro Tag (6.38)
- Inkonsistenz: Belegung pro Schwester (6.39)
- Inkonsistenz: Einschränkung (6.40)
- Inkonsistenz: Wunsch (6.41)

Auch wenn keine offensichtliche Inkonsistenz vorliegt, kann es sein, dass INTERDIP keine Lösung für ein vorliegendes Dienstplanproblem findet, das heißt, in angemessener Zeit keine findet. In diesem Fall hat der Benutzer meherere Möglichkeiten, doch noch einen Dienstplan zu bekommen:

1. Reduzierung der roten Wünsche (6.15)
2. Reduzierung der globalen Einschränkungen (6.28)
3. Erweiterung der Belegungsschranken. Sind die Belegungsschranken eng gesteckt, also bei einer konkreten Belegung der Abstand zwischen Unter- und Obergrenze gering, bleibt für die Belegung eines Plans wenig Spielraum, wodurch in angemessener Zeit keine Lösung gefunden wird. Durch eine Erweiterung dieses Spielraums bekommt man eine gute Chance, einen Dienstplan zu bekommen.
   (a) je Schwester für Phase 1 (6.20), Phase 2 (6.21) und Phase 3 (6.22)
   (b) je Tag für Phase 1 (6.23), Phase 2 (6.24) und Phase 3 (6.25)
4. Veränderung der Muster, je nachdem, in welcher Phase eine automatische Belegung nicht möglich war: Muster Phase 1 (6.29) , Muster Phase 2 (6.30) , Muster Phase 3 (6.31) . Die Veränderung der Muster braucht allerdings einige Erfahrung mit INTERDIP, insbesondere mit der konkreten Belegung von Dienstplänen (6.29.2).
5. Erhöhen der Zeitschranke. In besonderen Fällen kann es auch helfen, INTERDIP länger rechnen zu lassen, z.B. über Nacht. Die Erfahrung zeigt aber, dass wenn nach einigen Minuten keine Lösung gefunden wurde, die Aussicht, nach einigen Stunden eine zu finden, sehr gering ist.

Achtung: siehe auch Veränderte Rahmenbedingungen und bestehende Teillösung (6.2.5)

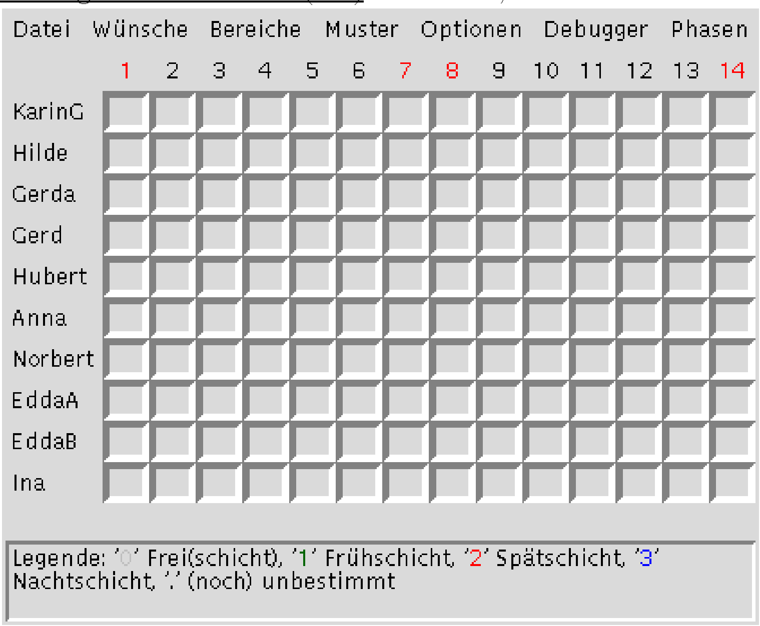
6.2.5 Veränderte Rahmenbedingungen und bestehende Teillösung

Werden Rahmenbedingungen (6.2.2) verändert, obwohl schon eine (Teil-) Lösung für den Dienstplan existiert, muss der Benutzer selbst dafür sorgen, die Phasen nochmal anzustoßen, auf die sich die Veränderung der Rahmenbedingungen auswirkt. Dafür existiert keine automatische Prüfung!

Als Hilfe kann der Benutzer nach der Änderung einer Rahmenbedingung die Bearbeitung eines Dienstplans (6.36) aufrufen, und dann den bestehenden auf Konsistenz prüfen. Wenn dann in einer Phase eine Inkonsistenz gemeldet wird, kann man davon ausgehen, dass diese und alle folgenden Phasen neu berechnet werden müssen.

6.3 Hauptfenster

Folgende Abbildung zeigt das Hauptfenster von INTERDIP (wie oben unter Benutzung des Handbuchs (6.1) erklärt ist, enthält das Bild Verweise):



Der Bildschirm besteht aus folgenden Bereichen:

- Menüleiste mit den Menüs
  - Datei (6.8) : Einstellungen laden und speichern, Dienstplan ausgeben, Programm beenden
  - Wünsche (6.9) : in rot, schwarz und weiß
  - Bereiche (6.10) : für die Belegung der Tage und der Schwestern und für globale Einschränkungen
  - Muster (6.11) : für die Belegung der Zellen des Dienstplans in den drei Phasen
  - Optionen (6.12)
  - Debugger (6.13) : damit kann unmittelbar in die Belegung des Dienstplans eingegriffen werden

  - Phasen (6.14) : Berechnung der Phasen

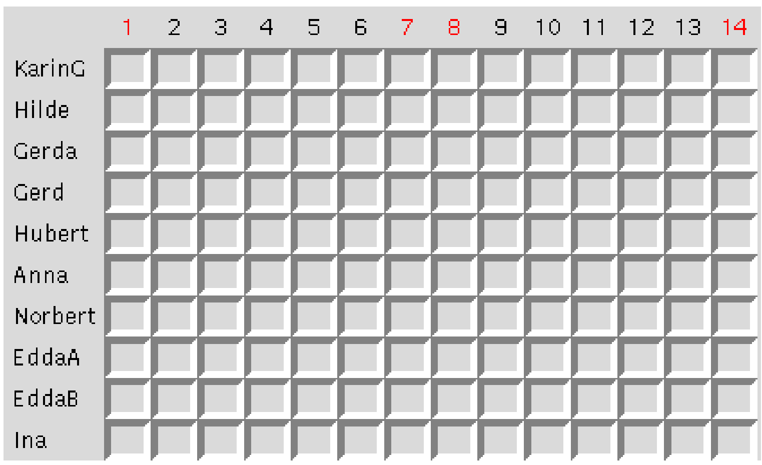
- Dienstplan (6.4): eine Tabelle mit einer Zeile für jede Schwester der Krankenstation und einer Spalte für jeden Tag des Planungszeitraums.

- Einer Statuszeile (6.6) zwischen Dienstplan und Legende

- Der Legende (6.7), die den Inhalt des ausgefüllten Dienstplans erklärt

Vom Hauptfenster aus, wird das ganze Programm bedient. Über die Menüs wird das Programm kontrolliert und im Dienstplan (6.4) wird die Planung angezeigt.

6.4 Dienstplan



Ein Dienstplan besteht aus

- der ersten Spalte links, die die Namen aller Schwestern trägt, für die der Dienstplan erstellt werden muss, das sind in der Regel alle Schwestern einer Krankenstation.

- der ersten Zeile oben, mit einer Zahl für jeden Tag des Planungszeitraums. In der Regel wird für einen Monat geplant, damit bezeichnen die Zahlen je eine Tag des Monats. Rote Zahlen bedeuten Feiertag, Samstag oder Sonntag, Tage also, an denen die meisten Leute frei haben wollen.

- der Tabelle mit Zellen für jeden Tag und jede Schwester. In diese Zellen trägt INTERDIP ein: ’1’ für Frühschicht, ’2’ für Spätschicht und ’3’ für Nachtschicht. Sind noch nicht alle Phasen berechnet, haben einige Zellen auch den Inhalt ’.’ für ¨noch unentschieden¨.

In diesem Feld des Hauptfensters (6.3) werden alle berechneten Pläne angezeigt und dem Benutzer zur Veränderung mit dem Debugger (6.13) zur Verfügung gestellt. Während einer Berechnung (6.2.3) wird immer der letzte generierte Dienstplan, das ist immer der bis dahin beste, hier angezeigt.

6.5 Die Schwestern-Spalte

KarinG
Hilde
Gerda
Gerd
Hubert
Anna
Norbert
EddaA
EddaB
Ina

Diese Spalte trägt die Namen aller Schwestern, für die der Dienstplan erstellt werden muss, das sind in der Regel die Namen aller Schwestern einer Krankenstation.

Übrigens: die Namen sind im Moment nicht vom Benutzer veränderbar.

6.6 Statuszeile

In der Statuszeile wird immer der aktuelle Zustand des Programms angezeigt. Im wesentlichen sind das:

- **phase n**
  es wird gerade Phase n berechnet
- **phase m phase n**
  der Benutzer hat die Phase m zur Berechnung angestoßen, es war aber die Phase n noch nicht berechnet (n=m-1), und INTERDIP hat Phase n automatisch angestoßen, die jetzt berechnet wird
- **BitteRücksprungpunktAnklicken...**
  Im Debugger hat der Benutzer den gezielten Rücksprung angeklickt. INTERDIP fordert den Benutzer jetzt auf, den Punkt (im Dienstplan) anzuklicken, zu dem zurückgesprungen werden soll.

6.7 Legende

Legende: '0' Frei(schicht), '1' Frühschicht, '2' Spätschicht, '3' Nachtschicht, '.' (noch) unbestimmt

Die meisten Fenster haben eine Legende. Diese erklärt immer in ein paar Worten, was das Fenster enthält. Zum Teil wird auch kurz erklärt, wie das Fenster zu bedienen ist.

Die Legenden können im Menü Optionen (6.12) ein- und ausgeschaltet werden.

Datei

6.8 Menü: Datei

Einstellungen laden
Einstellungen speichern
Dienstplan ausgeben
Dienstplan als LaTeX–Tabelle ausgeben
Ende

Im Menü Datei stehen folgende Befehle zur Verfügung:

- Einstellungen laden (6.47)
- Einstellungen speichern (6.47)
- **Dienstplan ausgeben**:
  der aktuell vollständigste und beste Dienstplan wird auf das Terminal, aus dem INTERDIP aufgerufen wurde, ausgegeben. Die Ausgabe als LaTeX-Tabelle kann so direkt in ein LaTeX-Dokument in eine figure-Umgebung eingefügt werden.
- **Ende**:
  INTERDIP wird beendet. ACHTUNG: Es findet keine Überprüfung statt, ob Einstellungen abgespeichert oder berechnete Dienstpläne ausgegeben wurden!

Wünsche

6.9 Menü: Wünsche

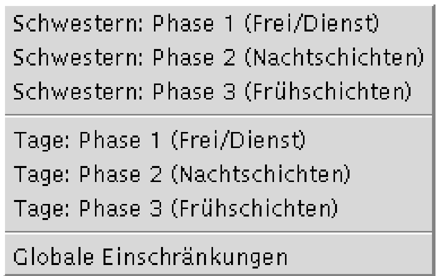
rot
schwarz
weiß

Im Menü Wünsche gelangt man zu den Eingaben für

- rote Wünsche (6.15) : unbedingt zu erfüllende Wünsche
- schwarze Wünsche (6.16): wichtige Wünsche
- weiße Wünsche (6.17): weniger wichtige Wünsche

Bereiche

6.10 Menü: Bereiche

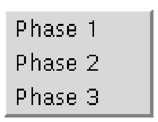


Im Menü Bereiche stehen folgende Befehle zur Verfügung:

- Festlegung der Belegungsgrenzen je Schwester für
  - Phase 1 (Frei/Dienst) (6.20)
  - Phase 2 (Nachtschichten) (6.21)
  - Phase 3 (Frühschichten) (6.22)
- Festlegung der Belegungsgrenzen je Tag für
  - Phase 1 (Frei/Dienst) (6.23)
  - Phase 2 (Nachtschichten) (6.24)
  - Phase 3 (Frühschichten) (6.25)
- Globale Einschränkungen (6.28)

Muster

6.11 Menü: Muster



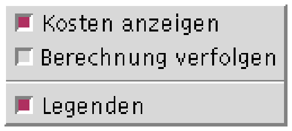
Hier gelangt man zu den Eingaben für die Belegungsmuster für jede Phase:

- Phase 1 (Frei/Dienst) (6.29)

- Phase 2 (Nachtschichten) (6.30)
- Phase 3 (Frühschichten) (6.31)

Optionen

6.12 Menü: Optionen

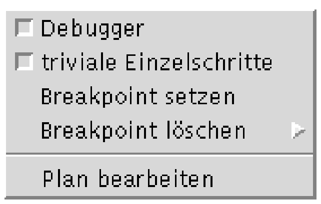


Hier werden generelle Optionen des Programms eingestellt:

- **Kosten anzeigen**
  Der Schalter für die Anzeige der Kosten (6.45).
- **Berechnung verfolgen**
  Ist dieser Schalter aktiviert, wird der Fortschritt der Berechnung einer Phase im Dienstplan (6.4) angezeigt: die Stelle, die gerade belegt wird.
- **Legenden**
  schaltet die Legenden (6.7) der Fenster ein bzw. aus.

Debugger

6.13 Menü: Debugger



Hier stehen die Befehle zur Verfügung, mittels derer der Benutzer unmittelbar in die Planung eingreifen kann:

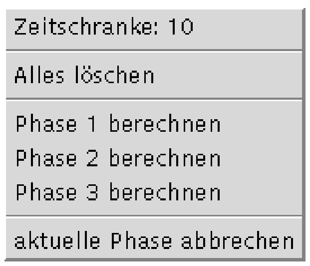
- Debugger (6.34)
  - **Debugger ein- und ausschalten** (geht nur, wenn gerade keine Berechnung stattfindet)
  - **triviale Einzelschritte** verfolgen: normalerweise (wenn dieser Schalter ausgeschaltet ist), werden im Einzelschrittmodus (s. Debugger (6.34)) die trivialen Schritte nicht angezeigt. Wenn der Schalter **triviale Einzelschritte** jedoch gesetzt ist, bekommt der Benutzer auch die Schritte angezeigt, in denen nur eine Auswahl möglich ist.

- **Breakpoint setzen:** hier kann der Benutzer eine Zelle des Dienstplans auswählen, an der die Berechnung (in jeder Phase) automatisch angehalten wird. Wird dieser Befehl aufgerufen, wartet INTERDIP, bis der Benutzer eine Zelle des Dienstplans angeklickt hat und trägt diese dann als Breakpoint ein. Die Breakpoints sind nur bei eingeschaltetem Debugger aktiv.
- **Breakpoint löschen:** hier können gesetzte Breakpoints wieder gelöscht werden. Dazu werden in einem Untermenü alle gesetzten Abbruchpunkte angezeigt (Schwester und Tag) und der davon ausgewählte wird gelöscht.

• Plan bearbeiten (6.36)Hiermit kann der Benutzer unmittelbar einen von INTERDIP generierten Plan bearbeiten und diesen dann auf Korrektheit testen.

Phasen

6.14 Menü: Phasen



In diesem Menü werden die Berechnungen angestoßen. Im einzelnen:

- **Zeitschranke**
  definiert die Zeitschranke, bei der automatische Planungen abgebrochen werden sollen. Der Wert wird im Fenster Wert ändern (6.46) geändert.

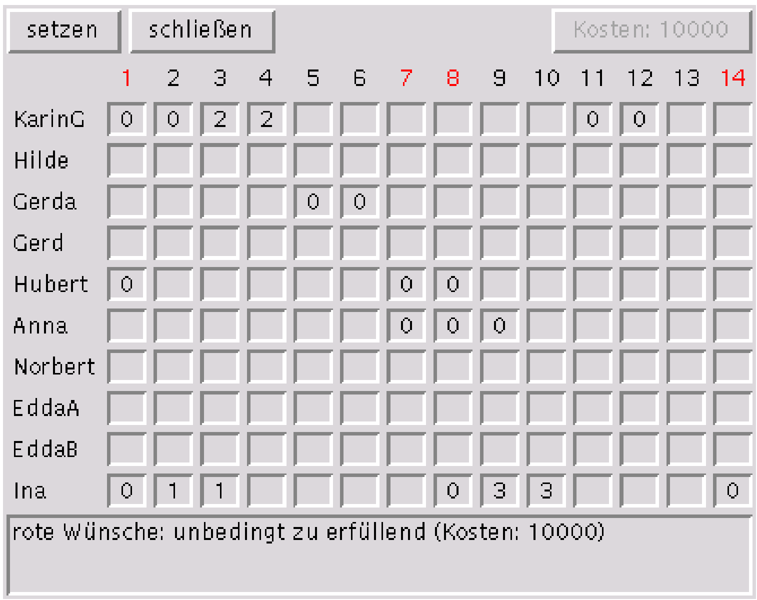
- **Alles löschen**
  Damit werden alle bereits berechneten Ergebnisse gelöscht. Wird dann eine Phase zur Berechnung angestoßen, werden müssen alle vorhergehenden Phasen neu berechnet werden.

- **Phase n berechnen**
  Damit kann der Benutzer die n-te Phase zur Berechnung anstoßen. Ist für die Phase n die Phase m=n-1 noch nicht berechnet worden, dann wird diese automatisch angestoßen. Was genau passiert, steht unter

  - Berechnen einer Phase (6.2.3)
  - Phase 1 (6.42)

– Phase 2 (6.43)
– Phase 3 (6.44)

6.15 Wünsche: rot



Die "roten" Wünsche sind die wichtigsten, sogar unbedingt zu erfüllenden Wünsche.

Dazu gehören neben den von den Schwestern als "sehr wichtig" geäußerten auch "Wünsche" wie Urlaub oder Mutterschaftsurlaub. Diese sind entweder gesetzlich vorgeschrieben (Mutterschaftsurlaub) oder aber werden getrennt vom eigentlichen Dienstplan geplant: Urlaub wird oft sehr viel früher als der konkrete Dienstplan vergeben und ist auch nicht automatisch von einem Dienstplanugsprogramm zu vergeben.

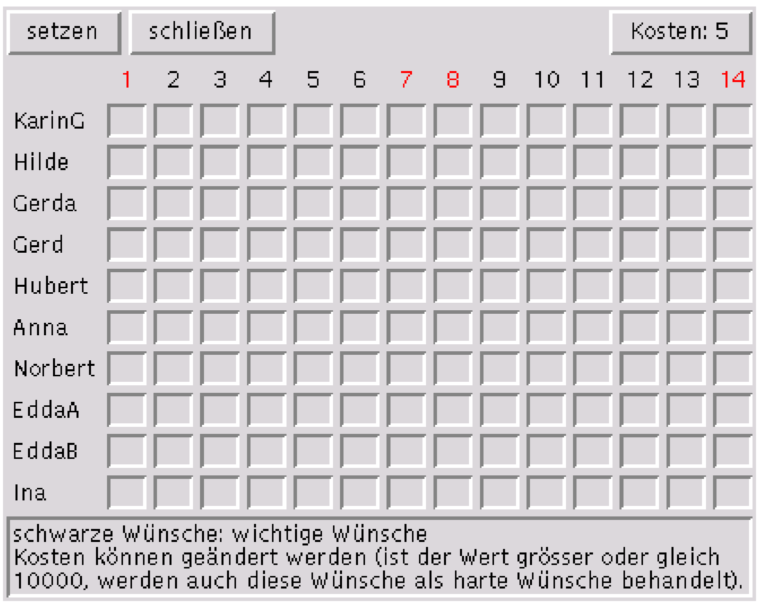
Indem diese Wünsche von jedem generierten Plan erfüllt werden müssen, kann es passieren, dass sie einen korrekten Plan verhindern. Wenn nämlich zum Beispiel zu viele Leute an einem Tag unbedingt frei haben wollen, kann es sein, dass die restlichen Schwestern nicht ausreichen, um die Station ausreichend zu besetzen. Dieses Problem kann von INTERDIP nicht gelöst werden. INTERDIP kann hier nur Hinweise geben, an welchen Stellen im Dienstplan solche Engpässe existieren, diese aber müssen vom Benutzer behoben werden, weil nur der entscheiden kann, welche der sehr wichtigen Wünsche tatsächlich weniger wichtig sind und in die Kategorie "schwarze Wünsche (6.16)" übernommen werden können.

Zur Bedienung des Fensters siehe Wünsche: Menü (6.18) und Wünsche: Tabelle (6.19) .

6.15.1 Zur Bezeichnung der Wünsche

Üblicherweise wird in einer Krankenstation ein Wunschplan ausgehängt, in den alle Schwestern der Station ihre Wünsche für einen Monat eintragen können, der noch nicht geplant ist. Die Wünsche können in zwei Farben eingetragen werden: "rot" für die sehr wichtigen und "weiß" für die weniger wichtigen Wünsche. Die dritte Kategorie der Wünsche in INTERDIP heißt "weiß", weil das die Wünsche sind, die quasi von vornherein auf jedem Wunschplan stehen (s. Wünsche: weiss (6.17)).

6.16 Wünsche: schwarz



Die schwarzen Wünsche sind die weniger wichtigen Wünsche. Zur Bezeichnung siehe Bezeichnung der Wünsche (6.15.1).

Die in dieser Kategorie erfassten Wünsche werden nicht unbedingt erfüllt. Sie dienen der Optimierung eines Dienstplans (6.16.1).

Zur Bedienung des Fensters siehe Wünsche: Menü (6.18) und Wünsche: Tabelle (6.19) .

6.16.1 Optimierung eines Dienstplans

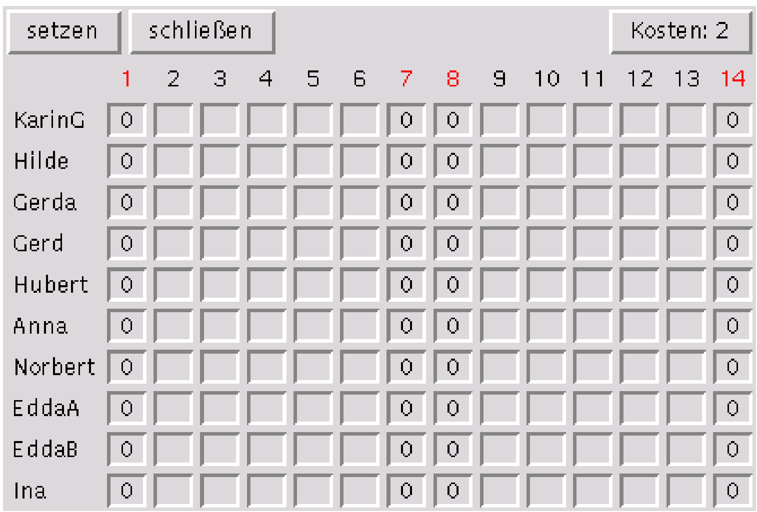
Ist in einer Phase der Berechnung eine Lösung gefunden, so versucht INTERDIP, diese zu verbessern. Dafür werden fiktive Kosten (6.45.1) erzeugt.

Die Kosten eines gesamten Dienstplans entstehen für jede Schwester getrennt (Die Kosten können auch angezeigt werden: Kosten anzeigen (6.12)). Die Optimierung versucht dann, Spitzen in den Kostenwerten abzubauen. Das heißt, von den Kosten eines Dienstplans, für jede Schwester getrennt, wird versucht, den maximalen zu verringern. Dadurch kommt keine Schwester besonders schlecht weg, was im allgemeinen für gerecht gehalten wird.

Ist der Benutzer mit einer eigentlich schon optimierten Lösung nicht zufrieden, kann er

- die den Kosten (6.45.1) zugrundeliegenden Werte verändern, wodurch die weichen Bedingungen anders gegeneinander gewichtet werden und damit stärker oder schwächer in die Optimierung des Dienstplans einfließen (siehe Kosten (6.45.1))
- während der Berechung einer Phase die Belegung des Dienstplans mit dem Debugger (6.34) beeinflussen
- eine (Teil-) Lösung selbst bearbeiten (6.36)

6.17 Wünsche: weiß



Diese Wünsche sind die am wenigsten wichtigen. Sie sind (wie gesagt (6.15.1)) die Wünsche, die auf jedem Wunschplan von vornehereien stehen.Das ist vor allem der Wunsch, am Wochenende und den Feiertagen frei zu haben.

Zur Bedienung des Fensters siehe Wünsche: Menü (6.18) und Wünsche: Tabelle (6.19) .

6.17.1 Behandlung der Wochenenden

Die Wochenenden spielen in INTERDIP an zwei Stellen eine Rolle. An obiger Abbildung sieht man, wie in INTERDIP die Freiwünsche gehandhabt werden: an den entsprechenden Tagen ist für jede Schwester der Wunsch "frei" eingetragen.

Die andere Stelle, die für die Wochenenden wichtig ist, sind die Muster für die Phase 1 (6.29). Dort sorgen entsprechende Muster für sinnvolle Schichtfolgen (Kombinationen von Schichten für eine Schwester an aufeinanderfolgenden Tagen). Das ¨Standardmuster¨ (siehe Benutzung der Muster (6.29.1)) definiert einen Arbeitsrhythmus ¨5 Tage arbeiten, 2 Tage frei¨ und eine entsprechende Fügung dieser Tage in den Wochenrhythmus des zu planenden Monats. Ist z.B. der erste des Monats ein Sonntag, so hat das Standardmuster folgendes Format: '0,.,.,.,.,.,0'. Ist der zweite ein Samstag und der dritte ein Sonntag, dann hat das Standardmuster das Format '.,0,0,.,.,.,.'.

In einem leeren Plan sind die ¨Wochenend-Frei-Wünsche¨ übrigens nicht automatisch eingetragen, sie müssen vielmehr vom Benutzer von Hand erfasst werden. Genauso muss bei den Mustern für die Phase 1 (6.29) das Standardmuster so angepasst werden, dass es für die konkrete Abfolge dieser Tage im Planungszeitraum passt.

6.18 Wünsche: Menü

Das Menü der Fenster für die Erfassung der Wünsche:
Drei Knöpfe:

- **setzen**
  trägt die angezeigten Wünsche (6.19) in INTERDIP ein; erst dann werden sie von INTERDIP zur Berechnung benutzt

- **schließen**
  trägt wie *setzen* die Wünsche ein und schließt das Fenster dann. Genauso wird verfahren, wenn der Benutzer das Fenster im Rahmen schließt.

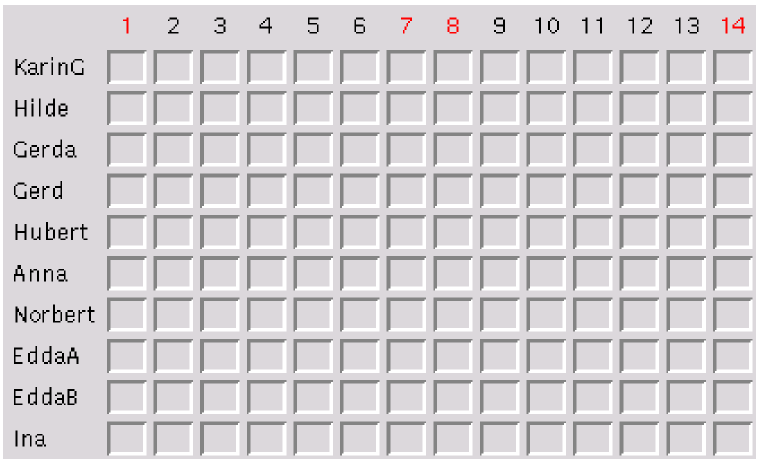
- **Kosten: k**
  hiermit können die Kosten für die Verletzung eines Wunschs der Kategorie festgelegt werden (siehe auch Wert ändern (6.46)).

6.18.1 Kosten für die Wünsche

Die roten Wünsche haben immer die Kosten *10000.* Das ist der Höchstwert und steht für ¨unbedingt zu erfüllend¨. Die Kosten für die roten Wünsche können also nicht verändert werden.

Die schwarzen Wünsche haben standardmäßig die Kosten 5 und die weißen die Kosten 2. Das sind konkret die Kosten, die entstehen, wenn ein Wunsch verletzt wird (s. Optimierung eines Dienstplans (6.16.1)). Diese können aber vom Benutzer verändert werden (siehe oben). Es kann sogar der Wert *10000* den schwarzen oder auch den weißen Wünschen zugeordnet werden, wodurch diese wie die roten behandelt werden. Der Benutzer kann also die Bedeutung der schwarzen und weißen Wünsche verändern.

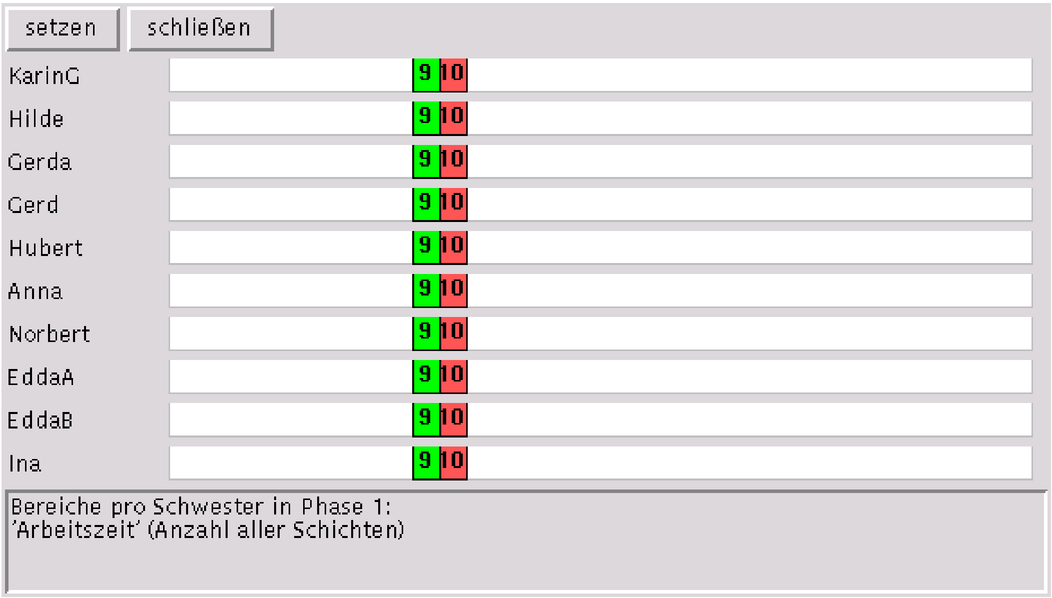
6.19 Wünsche: Tabelle



In die Tabelle eines Wünsche-Fensters werden die Wünsche eingegeben. Mit der Maus wird eine Zelle der Tabelle markiert, in die dann ein Eintrag gemacht werden kann. Aktuell mögliche Einträge sind:

- 1, 2 oder 3 für Früh-, Spät oder Nachtschicht
- 0 für einen Freiwunsch,
- d für eine Dienstschicht (also nicht frei), aber keine Nachtschicht,
- u für Urlaub (auch für Mutterschaftsurlaub u.ä.).
  Achtung: Urlaub darf nur als roter Wunsch angegeben werden; nur dann wird er korrekt verrechnet (für die Schwester als Arbeitszeit und für die Station als abwesend).

6.20 Bereiche je Schwester: Phase 1



Hier werden die Arbeitszeiten der Schwestern eingegeben.

Die Grenzen für die im Planungszeitraum zu leistenden Schichten sind für jede Schwester getrennt (mit Hilfe von Reglern (6.27)) anzugeben. Dabei ist es von Vorteil, einen gewissen Spielraum zu lassen, damit INTERDIP zur Planung hier Freiheiten hat. Als Faustregel gilt: Untergrenze ist die Regelarbeitszeit der Schwester und Obergrenze ist Untergrenze plus eine oder zwei Schichten, je nach Länge des Planungszeitraums.

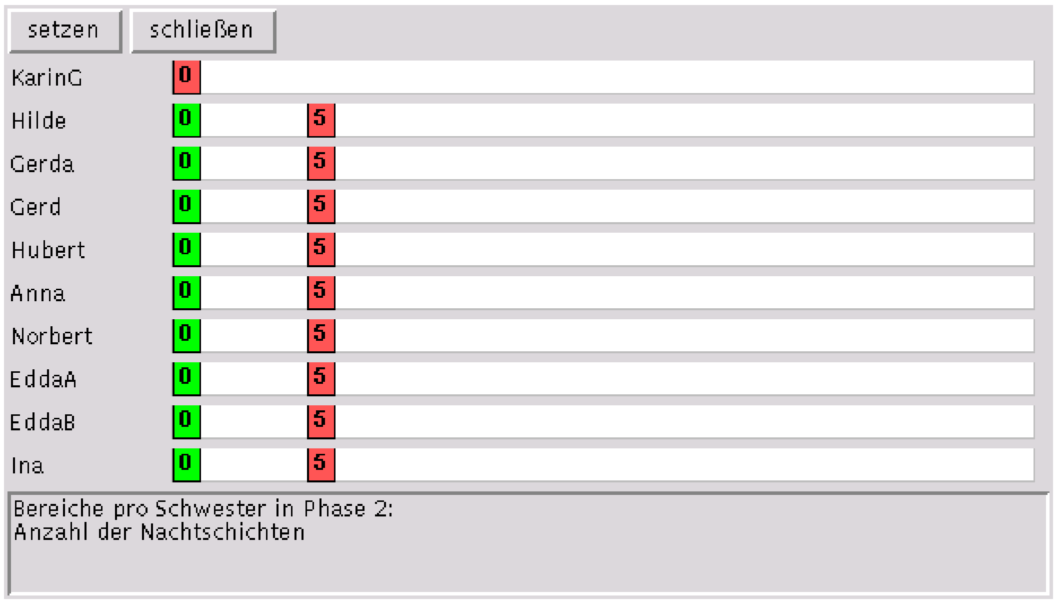
Die eingestellten Werte werden im Menü: setzen/schließen (6.26) in INTERDIP eingetragen.

6.20.1 Länge der Schichten; Schichten und Stunden

INTERDIP behandelt die Arbeitszeiten der Schwestern immer in ganzen Schichten und alle Schichten werden als gleich lang angenommen. Der Benutzer muss die im Planungszeitraum zu leistenden Stunden in Schichten umrechnen. Meist gilt: eine Schicht für acht Stunden. Die Anzahl der dadurch zu leistenden Schichten ist in Bereich je Schwester: Phase 1 (6.20) festzulegen.

Die Länge der Schichten wird also einheitlich angenommen. In der Praxis macht das jedoch kaum Probleme: die Schichten sind in der Regel maximal 10 Prozent voneinander verschieden. Ist also z.B. die Nachtschicht 10 Prozent länger als Früh- oder Spätschicht und bekommt eine Schwester in einem Monat 10 Nachtschichten zugeteilt (üblicherweise eher weniger) so hat sie am Ende des Monats Überstunden im Umfang einer Schicht.

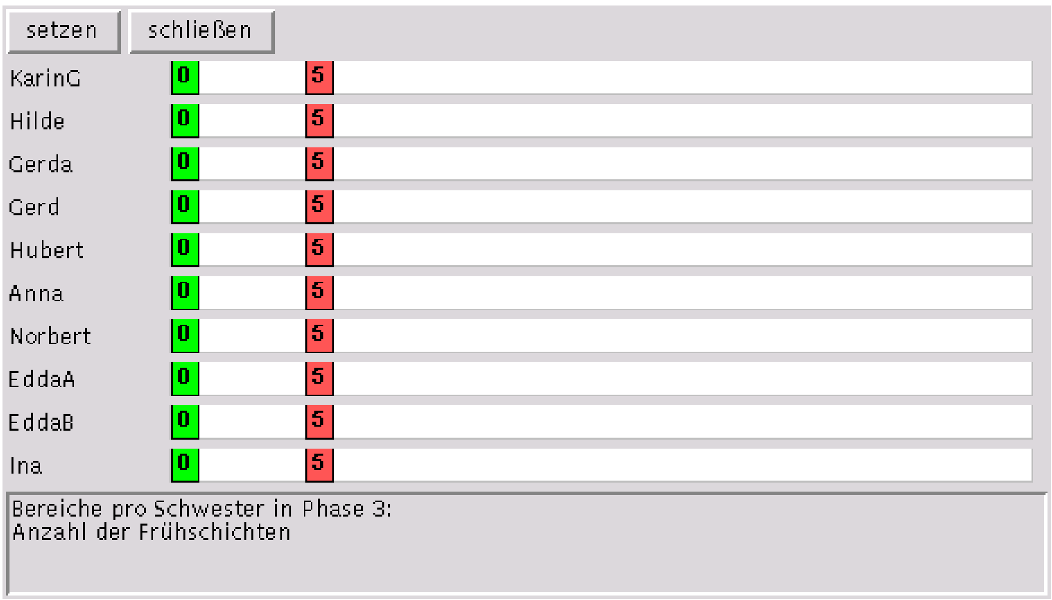
6.21 Bereiche je Schwester: Phase 2



Hier werden die Ober- und Untergrenzen für die Zuweisung von Nachtschichten pro Schwester festgelegt (mit Hilfe von Reglern (6.27)). Hier sollte INTERDIP einige Freiheit gelassen werden, allerdings nicht unbedingt von dem Umfang in gezeigter Abbildung.

Die eingestellten Werte werden im Menü: setzen/schließen (6.26) in INTERDIP eingetragen.

6.22 Bereiche je Schwester: Phase 3

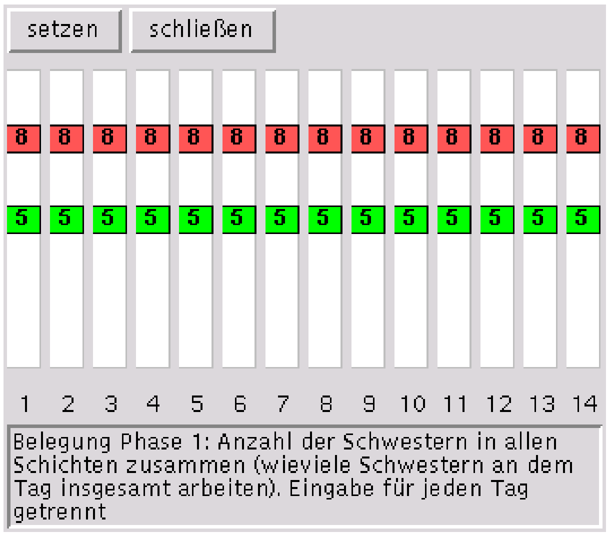


Hier werden die Ober- und Untergrenzen für die Zuweisung von Frühschichten pro Schwester festgelegt (mit Hilfe von Reglern (6.27)). Hier sollte INTERDIP einige Freiheit gelassen werden, allerdings nicht unbedingt von dem Umfang in gezeigter Abbildung.

Übrigens: für die Zuweisung von Spätschichten können in INTERDIP nicht unmittelbar Grenzen festgelegt werden. Diese ergeben sich aus implizit aus Arbeitszeit (s. Bereiche je Schwester: Phase 1 (6.20)), Nachtschichten (s. Bereiche je Schwester: Phase 2 (6.21))und Frühschichten.

Die eingestellten Werte werden im Menü: setzen/schließen (6.26) in INTERDIP eingetragen.

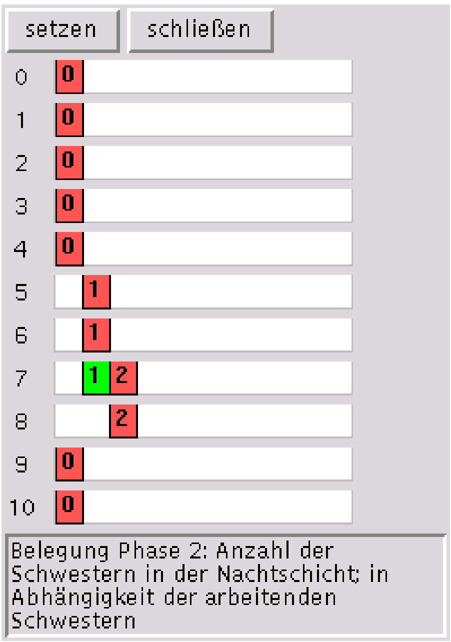
6.23 Belegungsgrenzen Phase 1



Hier werden die Belegungsgrenzen für die Phase eins festgelegt. Das ist die gesamte Belegung der Station über alle Schichten an einem Tag. Dies passiert individuell für jeden Tag (mit Reglern (6.27)). Ist wie hier für einen Tag als Untergrenze '5' und als Obergrenze '8' angegeben, heißt das, dass an diesem Tag der Station mindestens fünf und höchstens acht Schwestern zugeteilt werden, die sich dann auf alle drei Schichten verteilen (s. Belegungsgrenzen Phase 2 (6.24) und Belegungsgrenzen Phase 3 (6.25)).

Die eingestellten Werte werden im Menü: setzen/schließen (6.26) in INTERDIP eingetragen.

6.24 Belegungsgrenzen Phase 2

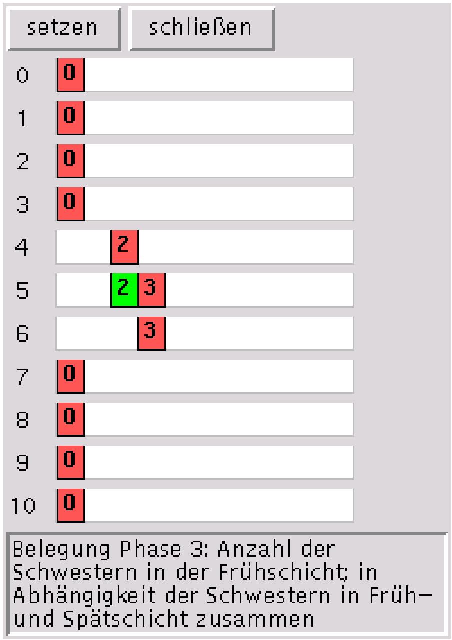


Hier kann der Benutzer die Belegungsgrenzen (pro Tag) für die zweite Phase angeben: die Nachtschichten. Die Werte werden mit Reglern (6.27) eingegeben. Die Zahlen links am Rand stehen für die Gesamtbelegung der Schicht an einem Tag. Hat INTERDIP also in der ersten Phase (6.42) für einen bestimmten Tag sechs Schwestern zur Arbeit eingeteilt (siehe auch Belegungsgrenzen Phase 1 (6.23)), so bedeuten die angezeigten Grenzen, dass an diesem Tag mindestens eine und höchstens zwei Schwestern in der Nachtschicht zu arbeiten haben.

Achtung: diese Grenzen werden nicht für jeden Tag einzeln angegeben!

Die eingestellten Werte werden im Menü: setzen/schließen (6.26) in INTERDIP eingetragen.

6.25 Belegungsgrenzen Phase 3



Hier werden die Belegungsgrenzen (pro Tag) für die dritte Phase angegeben (6.27): die Frühschichten. Die Bedeutung der Grenzen ist ähnlich denen in Phase 2 (6.24): stehen für einen bestimmten Tag noch n Schwestern zur Verfügung, ist also n Schwestern weder freigegeben noch eine Nachtschicht zugeteilt worden, so gilt als Belegungsunter- bzw. Obergrenze die grüne bzw. rote Zahl in Zeile n. Hier also: sind für Früh- und Spätschicht zusammen noch 4, 5 oder 6 Schwestern übrig, so sollen mindestens zwei und höchstens drei Schwestern in der Frühschicht, der Rest in der Spätschicht arbeiten (siehe auch Phase 3 (6.44)). Andere Möglichkeiten gibt es, resultierend aus den Belegungsgrenzen Phase 1 (6.23) und Belegungsgrenzen Phase 2 (6.24), nicht, deshalb ist es egal was in den anderen Zeilen steht (hier '0').

Die eingestellten Werte werden im Menü: setzen/schließen (6.26) in INTERDIP eingetragen.

6.26 Menü: Setzen/Schließen

Einige der Fenster haben zwei Knöpfe:

- **setzen**
  trägt die angezeigten Einstellungen in INTERDIP ein; erst dann werden sie von INTERDIP zur Berechnung benutzt

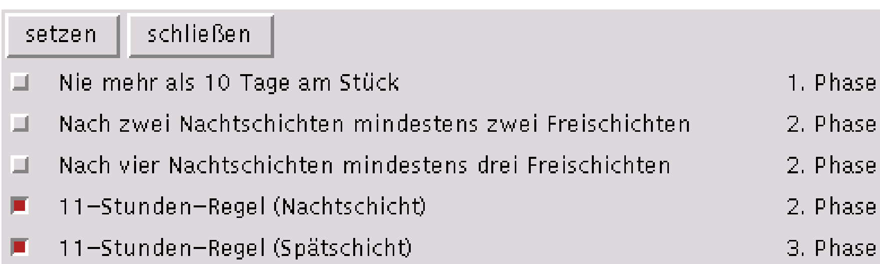
- **schließen**
  trägt wie *setzen* die Einstellungen ein und schließt das Fenster dann. Genauso wird verfahren, wenn der Benutzer das Fenster im Rahmen schließt.

Achtung: es gibt keine Möglichkeit, diese Fenster zu schließen, ohne die momentan angezeigten (und eventuell schon geänderten) Einstellungen in INTERDIP zu übernehmen!

6.27 Regler

Ein Regler wird zur Einstellung einer Ober- und einer Untergrenze verwendet. Die Untergrenze ist die grün hinterlegte Zahl und die Obergrenze die rot hinterlegte. Regler sind immer entweder waagerecht oder senkrecht angeordnet. Bei einem waagerechten Regler kann der rote Kasten nie links vom grünen sein, bei einem senkrechten nie darunter. Mit der Maus lassen sich die Kästchen verschieben. Zeigt ein Regler zufällig nur einen roten Kasten, ist der güne "dahinter verschwunden". Der wird wieder sichtbar, wenn der rote weggeschoben wird.

6.28 Globale Einschränkungen



Hier lassen sich die globalen Einschränkungen ein- und ausschalten. Ein farbig gefülltes Kästchen links vor dem Name der Einschränkung zeigt an, dass die Einschränkung eingeschaltet ist. Sie wird dann bei der Planung verwendet (siehe auch Konsistenz (6.2.4)). Rechts vom Namen der Einschränkung ist die Phase genannt, in der sich die Einschränkung auswirkt.

Folgende globale Einschränkungen sind definiert:

- **Nie mehr als zehn Tage am Stück**
  Keiner Schwester werden mehr als zehn Tage am Stück eine Arbeitsschicht zugewiesen.

- **Nach zwei Nachtschichten mindestens zwei Freischichten**
  Jede Schwester bekommt nach zwei Nachtschichten mindestens zwei freie Tage

- **Nach vier Nachtschichten mindestens drei Freischichten**
  Jede Schwester bekommt nach vier Nachtschichten mindestens drei freie Tage

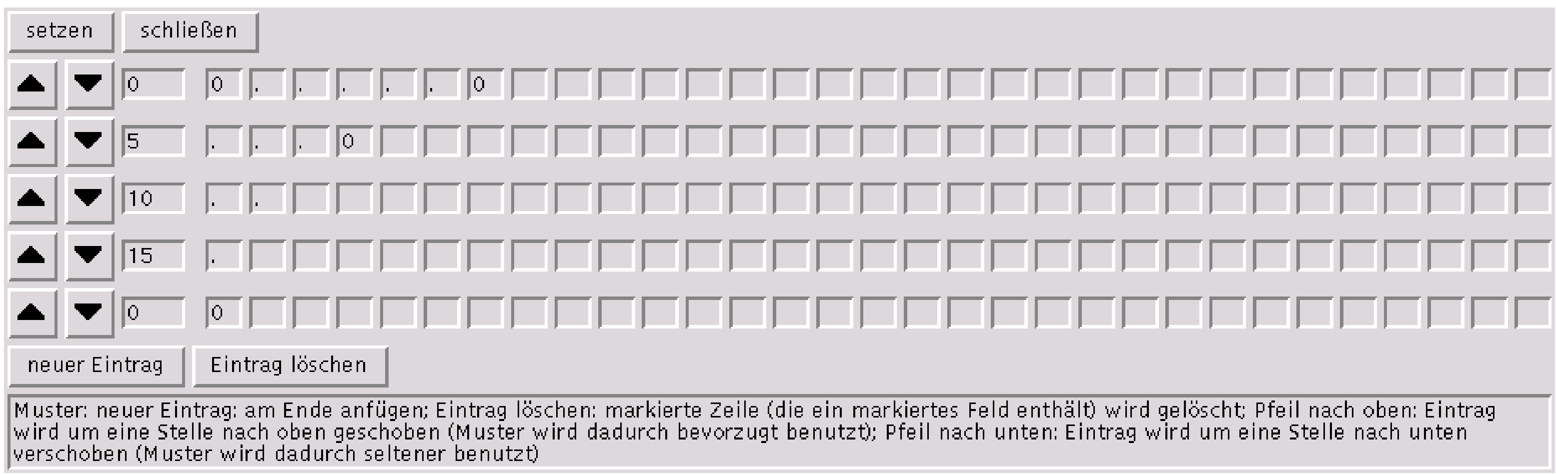
- **11-Stunden-Regel (Nachtschicht)**
  Muss eine Schwester an einem Tag in der Nachtschicht arbeiten, wird sie am nächsten Tag nur entweder wieder eine Nachtschicht oder aber eine Freischicht zugeteilt bekommen.

- **11-Stunden-Regel (Spätschicht)**
  Am Tag nach einer Spätschicht darf eine Schwester nicht in der Frühschicht arbeiten.

Alle diese Einschränkungen sind harte Bedingungen!

Die eingestellten Werte werden im <u>Menü: setzen/schließen (6.26)</u> in INTERDIP eingetragen.

6.29 Muster Phase 1



Die Bedienung dieses Fensters erfolgt über die darin enthaltenen Elemente:

- <u>Menü: setzen/schließen (6.26)</u>

- mehreren <u>Muster-Zeilen (6.32)</u> mit Einträgen

- <u>Menü: neuer Eintrag / Eintrag löschen (6.33)</u>

Die Muster für die Phase 1 (6.42) geben sinnvolle Kombinationen von Arbeits- und Freischichten an. Diese werden in der Reihenfolge von oben nach unten zur Belegung des Dienstplans benutzt. Eine sinnvolle Kombination ist z.B. "fünf Tage arbeiten, zwei Tage frei". Das ist auch das erste Muster der Abbildung (siehe auchBehandlung der Wochenenden (6.17.1)). Später kommen die trivialen Muster (siehe unten).

6.29.1 Benutzung der Muster

Die Muster werden zur Belegung des Dienstplans in der jeweiligen Phase benutzt (siehe auch Muster Phase 1 (6.29), Muster Phase 2 (6.30) und Muster Phase 3 (6.31)). Dabei werden Muster in der Reihenfolge von oben nach unten zur Belegung (6.29.2) verwendet. Das erste der Liste heißt deshalb auch **Standardmuster**. Ein Muster besteht aus Einträgen in den Dienstplan und Kosten für die Verwendung. Wird ein Muster verwendet, werden die entsprechenden Einträge vorgenommen und die Kosten dafür zu den Kosten der entsprechenden Schwester addiert (siehe auch Optimierung eines Dienstplans (6.16.1)). Eine Liste von Mustern sollte neben, das heißt in der Reihenfolge nach, sinnvollen Kombinationen auch triviale Kombinationen enthalten, damit ein Dienstplan, wo die sinnvollen nicht passen, wenigstens mit den trivialen aufgefüllt werden kann, damit überhaupt eine Lösung gefunden wird. Die trivialen sind die Muster, die nur einen Eintrag haben. Außerdem müssen unter den trivialen Mustern für eine Schicht auch solche mit Einträgen stehen, die in früheren Schichten gemacht wurden, damit die Berechnung einer Phase diese Belegungen erhalten kann (siehe z.B. Muster Phase 3 (6.31)).

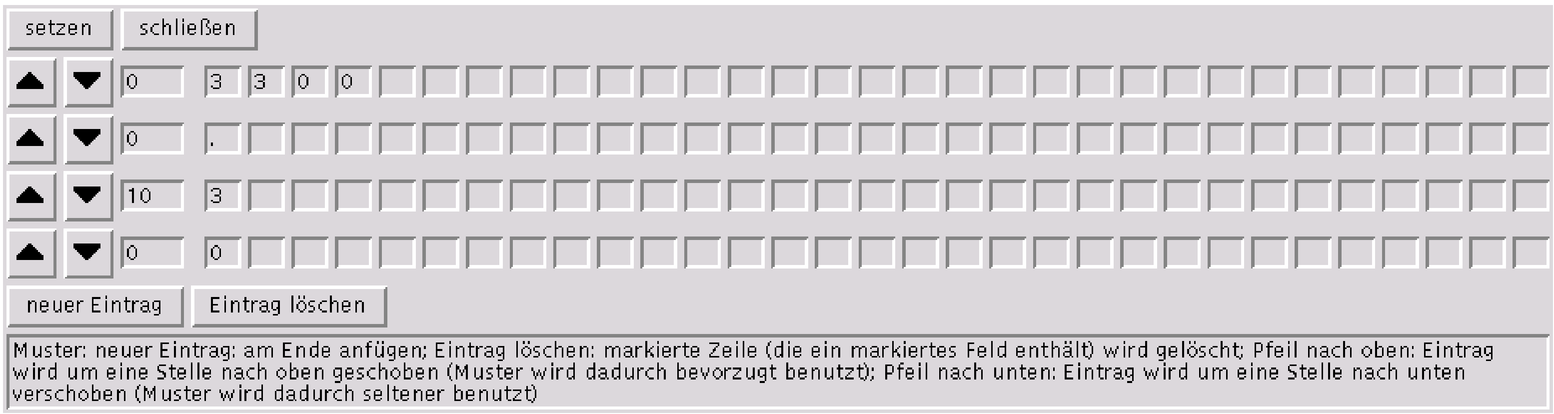
6.29.2 Belegung des Dienstplans

Vor allem zum besseren Verständnis der Muster soll hier kurz die Belegung des Dienstplans beschrieben werden. Die Belegung eines Dienstplans ist ein Teil einer Phase (6.2.3).

Die Belegung eines Dienstplans (6.4) passiert immer von links oben nach rechts unten, und dabei zuerst von links nach rechts und dann von oben nach unten. Es wird also zeilenweise belegt und eine Zeile von links nach rechts. An einer beliebigen Stelle des Dienstplans wird zuerst nach einem hier passenden Muster gesucht, wobei die Muster der Phase in der Reihenfolge von oben nach unten probiert werden. In der Regel ist ein Plan schon durch Wünsche (6.15) vorbelegt, oder durch andere Bedingungen eingeschränkt, so dass nicht an jeder Stelle jedes Muster passt. Das erste passende Muster wird dann verwendet und in den Dienstplan eingetragen und die Kosten (6.16.1) zu denen der Schwester, deren Zeile gerade belegt wird, hinzuaddiert. Dann wird für die nächste Zelle, die also an die zuletzt belegte Zelle anschließt, mit der Belegung fortgefahren.

Muster werden also an jeder beliebigen Stelle des Dienstplans verwendet, was vor allem für das Standardmuster der ersten Phase einige Bedeutung hat: siehe Behandlung der Wochenenden (6.17.1). Außerdem werden Belegungen nur nach vorhandenen Mustern vorgenommen, wodurch triviale Muster notwendig sind: siehe Benutzung der Muster (6.29.1).

6.30 Muster Phase 2

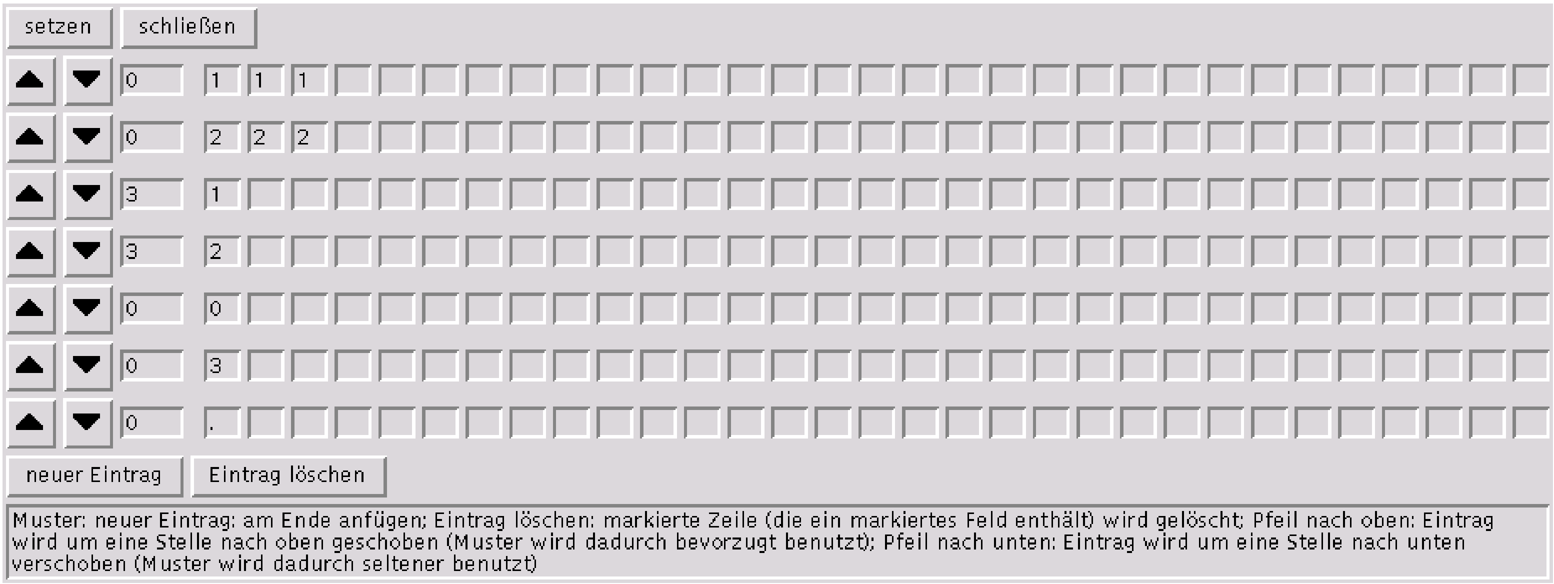


Die Bedienung dieses Fensters erfolgt über die darin enthaltenen Elemente:

- Menü: setzen/schließen (6.26)
- mehreren Muster-Zeilen (6.32) mit Einträgen
- Menü: neuer Eintrag / Eintrag löschen (6.33)

Die Muster der Phase 2 (6.43) geben sinnvolle Kombinationen von Nachtschichten bzw. deren Einbettung in Folgen von Arbeits- und Freischichten an. Die Muster der Abbildung bedeuten z.B.: am besten ist es, zwei Nachtschichten nacheinander zu belegen, aber nur, wenn anschliessend zwei Tage frei sind (die sind ja durch Phase 1 (6.42) schon festgelegt). Wo das nicht möglich ist, wird zunächst keine Nachtschicht belegt (die Zelle bleibt unbestimmt) und erst dann eine einzelne Nachtschicht vergeben, wofür Kosten von 10 fällig werden. Das letzte Muster ist das triviale Muster (siehe Benutzung der Muster (6.29.1)) für Freischichten aus der ersten Phase.

6.31 Muster Phase 3



Die Bedienung dieses Fensters erfolgt über die darin enthaltenen Elemente:

- Menü: setzen/schließen (6.26)
- mehreren Muster-Zeilen (6.32) mit Einträgen
- Menü: neuer Eintrag / Eintrag löschen (6.33)

Die Muster der Phase 3 (6.44) geben sinnvolle Kombinationen von Früh- und Spätschichten an bzw. deren Einbettung in Frei- und Nachtschichten aus Phase 1 (6.42) und Phase 2 (6.43). Die Muster der Abbildung bedeuten: es ist immer besser, gleiche Schichten nacheinander zu belegen, als einzelne. Also werden zunächst drei gleich versucht. Dann einzelne, wofür Kosten entstehen. Die Muster am Ende der Liste sind die trivialen Muster (siehe Benutzung der Muster (6.29.1)).

6.32 Musterzeile

Die Musterzeile ist Teil der Fenster zur Eingabe phasenspezifischer Muster. Diese Zeile enthält folgende Elemente:

- **Pfeil nach oben (**  **)**
  dieser Eintrag wird um eine Stelle in der Liste nach oben geschoben. Dadurch wird dieses Muster häufiger benutzt. Siehe auch Benutzung der Muster (6.29.1) .

- **Pfeil nach unten ( )**
  dieser Eintrag wird um eine Stelle in der Liste nach unten geschoben. Dadurch wird dieses Muster weniger häufig benutzt. Siehe auch Benutzung der Muster (6.29.1) .

- **Kosten-Feld**
  hier wird der Wert für die Kosten angegeben, der bei Verwendung dieses Musters anfallen (siehe auch Optimierung eines Dienstplans (6.16.1)).

- **Alle folgenden Einträge** definieren das Muster. Ein Muster endet vor dem ersten leeren Eintrag. Enthält kein Feld einen Eintrag, wird dieses Muster gelöscht. Zu den möglichen Einträgen, siehe Muster Phase 1 (6.29) , Muster Phase 2 (6.30) und Muster Phase 3 (6.31) .

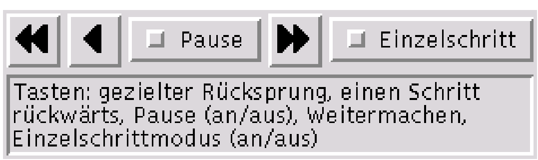
6.33 Muster-Menü: neuer Eintrag / Eintrag löschen

Mit diesen Knöpfen werden neue Muster erstellt oder bestehende gelöscht:

- **neuer Eintrag**
  fügt am Ende der Liste der Muster eine neue Zeile an

- **Eintrag löschen**
  löscht den gerade markierten Eintrag (das ist eine Zeile). Eine Zeile ist markiert, indem ein Feld der Zeile markiert ist. Dieses ist mit einem Rahmen versehen.

6.34 Debugger



Mit dem Debugger kann der Benutzer direkt in die Belegung eines Dienstplans eingreifen. Der Debugger wird im Menü: Debugger (6.13) eingeschaltet. Im Debug-Modus wird die Belegung eines Dienstplans im

Dienstplan (6.4) angezeigt, das heißt, es werden alle konkreten Belegungen sofort für den Benutzer sichtbar gemacht. Dieser kann die Belegung anhalten (Pause), gezielt an einen Punkt des Dienstplans und damit der Belegung zurückspringen, einen Schritt rückgängig machen, die Belegung fortsetzen oder den Einzelschrittmodus einschalten. Wird gezielt zurückgesprungen oder ein Schritt rückgängig gemacht, bekommt der Benutzer eine Auswahl (6.35) angeboten, aus der er selbst das Muster auswählt, das an der aktuellen Belegungsposition benutzt werden soll. Im Einzelschrittmodus wählt der Benutzer jeden Schritt selbst aus.

Folgende Befehle stehen im Debugger zur Verfügung:

- **gezielter Rücksprung ( )**
  der Benutzer wird durch eine Anzeige in der Statuszeile (6.6) aufgefordert, eine Stelle im Dienstplan (6.4) auszuwählen, das heißt anzuklicken, bis zu der die aktuelle Belegung rückgängig gemacht wird. Dort (diese Position ist orangefarben markiert) wählt (6.35) der Benutzer selbst eine Belegung aus.

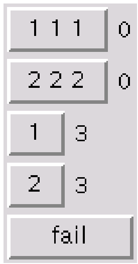
- **ein Schritt zurück ( )**
  von der augenblicklichen Position der Belegung (weiß markiert) aus wird ein Schritt, das ist die Belegung des letzten Musters, rückgängig gemacht.

- **Pause**
  hält die Belegung an. Im Debug-Modus wird nach der kompletten Belegung eines Dienstplans automatisch die Pause aktiviert, damit der Benutzer Gelegenheit hat, Teile der Belegung rückgängig zu machen und selbst zu beienflussen. Das heißt auch, dass im Debug-Modus keine Optimierung (6.16.1) stattfindet!

- **Belegung fortsetzen ( )**
  setzt die Belegung automatisch fort. Das heißt, die Pause und der Einzelschrittmodus werden ausgeschalten und falls der Benutzer gerade eine Auswahl zu treffen hat, wird auch die automatisch vorgenommen.

- **Einzelschrittmodus**
  Im Einzelschrittmodus hat der Benutzer an jeder Stelle die Auswahl (6.35) selbst vorzunehmen. Allerdings werden triviale Schritte nur zur Auswahl angeboten, wenn der entsprechende Schalter im Menü: Debugger (6.13) eingeschaltet ist (siehe Auswahl (6.35)).

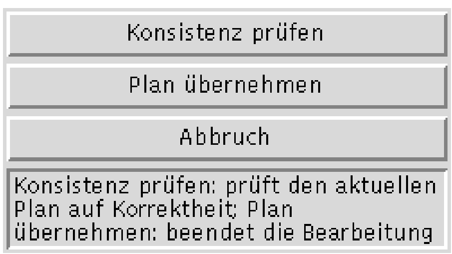
6.35 Auswahl



Ein solches oder ähnliches Fenster wird dem Benutzer angezeigt, wenn er im Debug-Modus (6.34) eine Auswahl zu treffen hat. Hier hat er für jede (noch) mögliche Belegung einen Knopf. Der Knopf ist mit der Belegung bezeichnet, die INTERDIP vornehmen soll. Neben dem Knopf stehen die Kosten, die für die Verwendung dieses Musters (6.29.1) entstehen. Der Knopf *fail* lässt die Belegung an dieser Stelle fehlschlagen und an der vorhergehenden Stelle neu versuchen.

Es werden also nur Muster angezeigt, die auch tatsächlich belegt werden können. Außerdem wird nach Rückverfolgung (oder *Backtracking*), wenn also der Benutzer an einer Stelle eine Auswahl getroffen hat, die Belegung aber wieder an diese Stelle zurückkommt, nur noch die Liste der Muster angeboten, die noch nicht versucht wurden. Das heißt, die Auswahlliste an einer Stelle wird durch Backtracking immer kleiner. Ist nur (noch) eine Auswahl möglich wird standardmäßig diese nicht vom Benutzer verlangt. Das kann mit dem Schalter triviale Einzelschritte (6.13) geändert werden.

6.36 Plan bearbeiten



Diese Möglichkeit steht im Menü: Debugger (6.13) zur Verfügung. Hiermit kann der Benutzer einen bestehenden Plan, egal welcher Phase, selbst bearbeiten. Dazu kann er in die Zellen des Dienstplans (6.4) Einträge machen bzw. Einträge verändern. Einen so bearbeiteten Dienstplan kann er dann auf Konsistenz, das ist Korrektheit bezüglich aller Rahmenbedingungen, überprüfen und nach INTERDIP übernehmen.

Folgende Befehle stehen hier zur Verfügung:

- **Konsistenz prüfen**

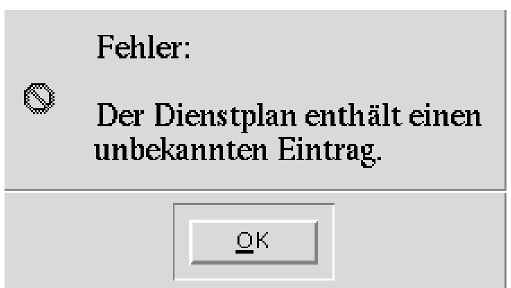
überprüft die Konsistenz des bearbeiteten Dienstplans. Folgende Ungültigkeiten werden gemeldet:

- Fehler: unbekannter Eintrag (6.37)
- Inkonsistenz: Belegung pro Tag (6.38)
- Inkonsistenz: Belegung pro Schwester (6.39)
- Inkonsistenz: Einschränkung (6.40)
- Inkonsistenz: Wunsch (6.41)

- **Plan übernehmen**
Hiermit wird der angezeigt Plan in INTERDIP übernommen. Darauf aufbauend können dann noch fehlende Phasen angestoßen werden. Die Bearbeitung ist damit beendet.

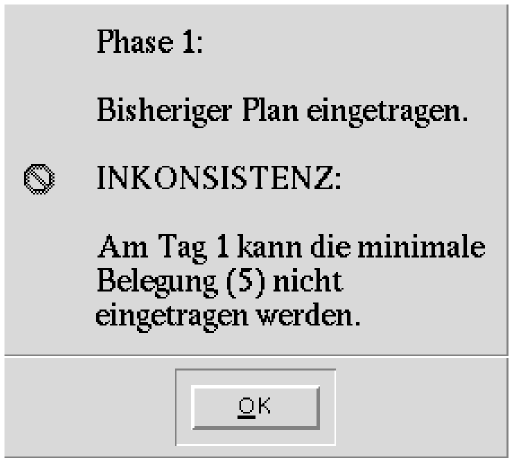
- **Abbruch**
beendet die Bearbeitung und macht alle Änderungen rückgängig.

6.37 Fehler: unbekannter Eintrag



Wurde vom Benutzer beim Bearbeiten des Dienstplans (6.36) ein ungültiger Eintrag vorgenommen, erscheint diese Meldung. Ein ungültiger Eintrag ist keine Inkonsistenz (6.2.4), sondern ein Eintrag, der in einem Dienstplan nie auftaucht, wie z.B. 'x'.

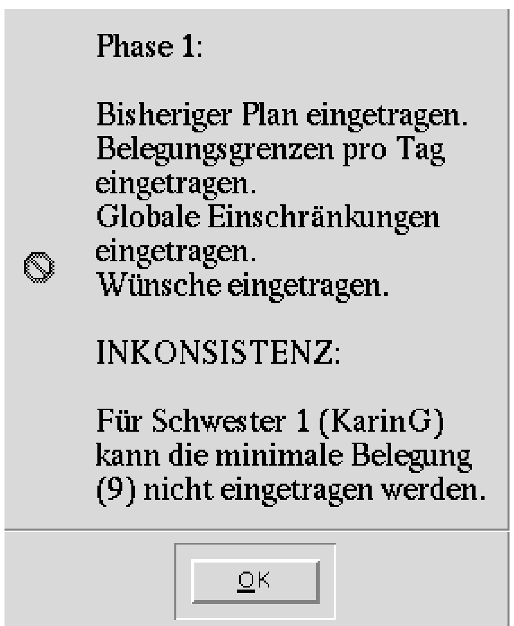
6.38 Inkonsistenz: Belegung pro Tag



Die minimale oder maximale Belegung pro Tag kann in einer Phase für den angegebenen Tag nicht eingehalten werden. In diesem Fall müssen die entsprechenden Grenzen angepasst werden, je nach Phase in: Belegung je Tag für Phase 1 (6.23), Phase 2 (6.24) und Phase 3 (6.25). Ist die Inkonsistenz für einen bearbeiteten Dienstplan (6.36) festgestellt worden, so können auch am angegebenen Tag die Grenzen durch Veränderung der Einträge eingehalten werden.

Siehe auch Konsistenz (6.2.4) .

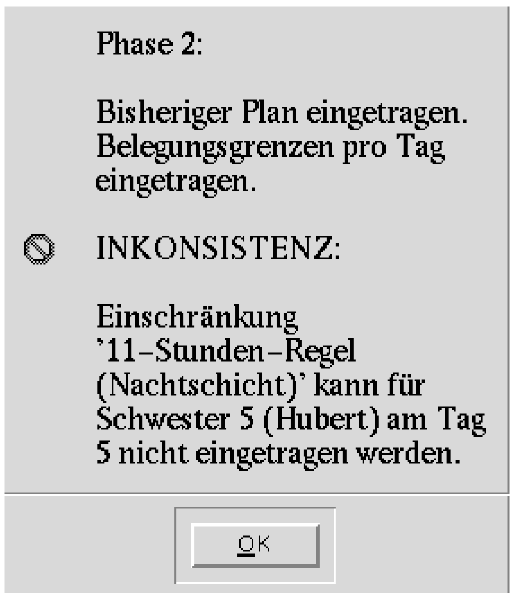
6.39 Inkonsistenz: Belegung pro Schwester



Die minimale oder maximale Belegung pro Schwester kann in einer Phase für die angegebene Schwester nicht eingehalten werden. In diesem Fall müssen die entsprechenden Grenzen angepasst werden, je nach Phase in: Belegung je Schwester für Phase 1 (6.20), Phase 2 (6.21) und Phase 3 (6.22). Ist die Inkonsistenz für einen bearbeiteten Dienstplan (6.36) festgestellt worden, so können auch für die angegebene Schwester die Grenzen durch Veränderung der Einträge eingehalten werden.

Siehe auch Konsistenz (6.2.4) .

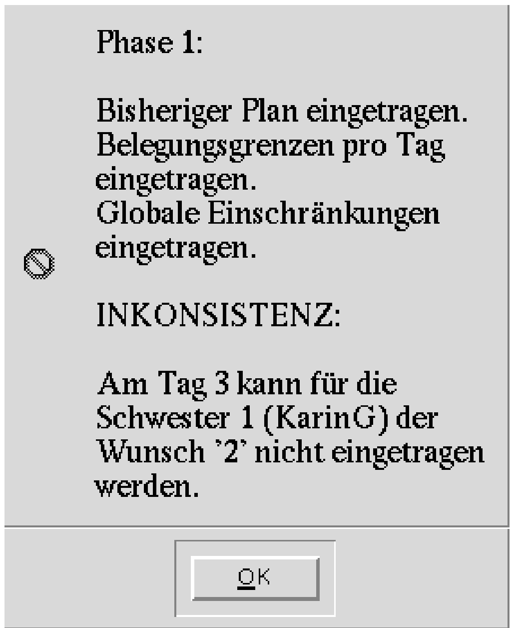
6.40 Inkonsistenz: Einschränkung



Die angegebene globale Einschränkung kann an der angegebenen Stelle des Dienstplans (6.4) nicht eingehalten werden. Sie muss gegebenenfalls bei den globalen Einschränkungen (6.28) ausgeschaltet werden. Alternativ kann versucht werden, durch Neuberechnung (6.2.3) der dieser Phase vorausgehenden und entsprechende Veränderung der Rahmenbedingungen (6.2.2) die Einschränkung in dieser Phase einzuhalten. Was hier *entsprechend* heißt, muss die Erfahrung zeigen. Ist die Inkonsistenz für einen bearbeiteten Dienstplan (6.36) festgestellt worden, so kann auch an der angegebenen Stelle die Einschränkung durch Veränderung des Eintrags eingehalten werden.

Siehe auch Konsistenz (6.2.4) .

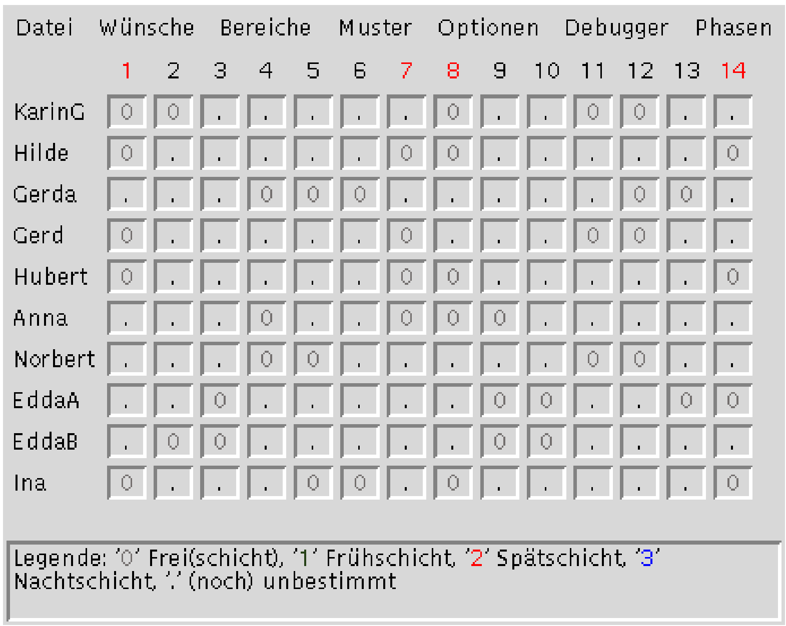
6.41 Inkonsistenz: Wunsch



Der angegebene harte Wunsch kann an der angegebenen Stelle des Dienstplans (6.4) nicht eingehalten werden. Der Wunsch muss gegebenenfalls entfernt werden. Meist ist das ein roter Wunsch (6.15). Je nach den Kosten für die Wünsche (6.18.1) kann das aber auch ein schwarzer Wunsch (6.16) oder ein weißer Wunsch (6.17) sein. Alternativ kann versucht werden, durch Neuberechnung (6.2.3) der dieser Phase vorausgehenden und entsprechende Veränderung der Rahmenbedingungen (6.2.2)den Wunsch in dieser Phase einzuhalten. Was hier *entsprechend* heisst, muss die Erfahrung zeigen. Ist die Inkonsistenz für einen bearbeiteten Dienstplan (6.36) festgestellt worden, so kann auch an der angegebenen Stelle der Wunsch durch Veränderung des Eintrags eingehalten werden. Grund für die Inkonsistenz kann auch ein trivial ungültiger Wunsch wie z.B. 'x' sein.

Siehe auch Konsistenz (6.2.4) .

6.42 Phase 1

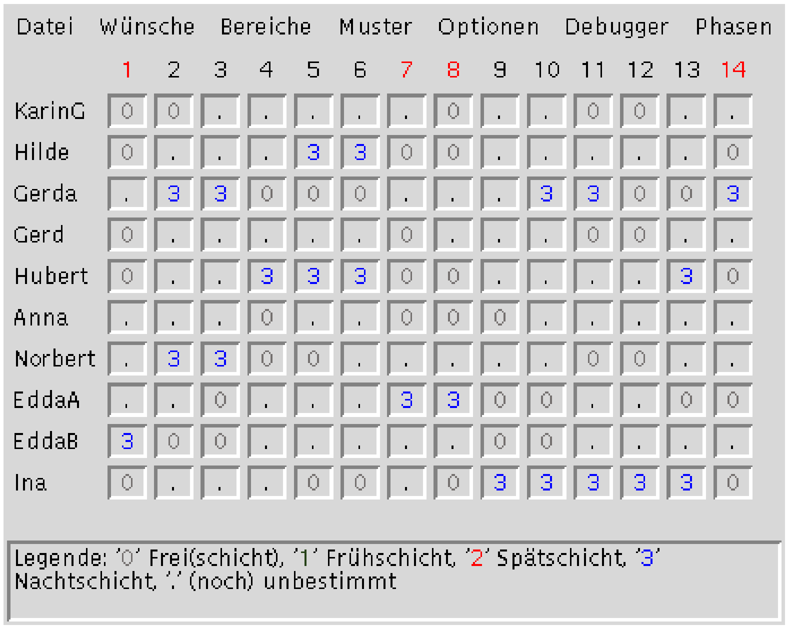


In Phase 1 werden die Freischichten bestimmt. Obige Abbildung zeigt einen Teilplan nach Beendigung der ersten Phase. Ein Eintrag '0' bezeichnet eine Freischicht, ein Eintrag '.' bedeutet, diese Zelle ist noch unbestimmt. Nachdem in den folgenden Phasen nur Nacht-, Früh- und Spätschichten belegt werden, und nach der letzten Phase alle Zellen einen bestimmten Eintrag haben, bedeutet '.' hier gleichzeitig eine Arbeitsschicht.

Wichtig: Freischichten werden nur in der ersten Phase belegt!

Für die Belegung des Plans in der ersten Phase werden alle entsprechenden Rahmenbedingungen verwendet: Wünsche: rot (6.15), Wünsche schwarz (6.16), Wünsche: weiß (6.17), Belegung pro Schwester (6.20), Belegung pro Tag (6.23), globale Einschränkungen (6.28) und Muster (6.29).

6.43 Phase 2



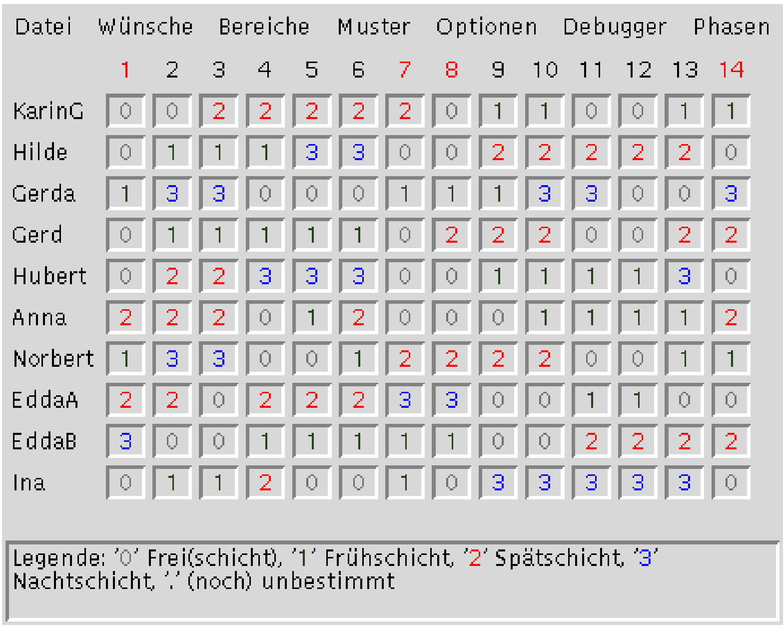
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
KarinG	0	0	.	.	.	.	.	0	.	.	0	0	.	.
Hilde	0	.	.	.	3	3	0	0	.	.	.	.	.	0
Gerda	.	3	3	0	0	0	.	.	.	3	3	0	0	3
Gerd	0	.	.	.	.	.	0	.	.	.	0	0	.	.
Hubert	0	.	.	3	3	3	0	0	.	.	.	.	3	0
Anna	.	.	.	0	.	.	0	0	0	.	.	.	.	.
Norbert	.	3	3	0	0	.	.	.	.	.	0	0	.	.
EddaA	.	.	0	.	.	.	3	3	0	0	.	.	0	0
EddaB	3	0	0	.	.	.	.	.	0	0	.	.	.	.
Ina	0	.	.	.	0	0	.	0	3	3	3	3	3	0

In der zweiten Phase werden die Nachtschichten belegt. Obige Abbildung zeigt einen Plan nach der zweiten Phase. Hier bedeuten wieder '0' Freischicht, '.' unbestimmt und '3' Nachtschicht.

Wichtig: Nachtschichten werden nur in der zweiten Phase belegt!

Für die Belegung des Plans in der zweiten Phase werden alle entsprechenden Rahmenbedingungen verwendet: Wünsche: rot (6.15), Wünsche schwarz (6.16), Wünsche: weiß (6.17), Belegung pro Schwester (6.21), Belegung pro Tag (6.24), globale Einschränkungen (6.28) und Muster (6.30).

6.44 Phase 3



Datei Wünsche Bereiche Muster Optionen Debugger Phasen

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
KarinG	0	0	2	2	2	2	2	0	1	1	0	0	1	1
Hilde	0	1	1	1	3	3	0	0	2	2	2	2	2	0
Gerda	1	3	3	0	0	0	1	1	1	3	3	0	0	3
Gerd	0	1	1	1	1	1	0	2	2	2	0	0	2	2
Hubert	0	2	2	3	3	3	0	0	1	1	1	1	3	0
Anna	2	2	2	0	1	2	0	0	0	1	1	1	1	2
Norbert	1	3	3	0	0	1	2	2	2	2	0	0	1	1
EddaA	2	2	0	2	2	2	3	3	0	0	1	1	0	0
EddaB	3	0	0	1	1	1	1	1	0	0	2	2	2	2
Ina	0	1	1	2	0	0	1	0	3	3	3	3	3	0

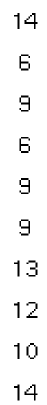
Legende: '0' Frei(schicht), '1' Frühschicht, '2' Spätschicht, '3' Nachtschicht, '.' (noch) unbestimmt

In der dritten Phase werden die Früh- und die Spätschichten belegt. Genaugenommen werden die Frühschichten belegt und was übrig bleibt, ist Spätschicht. Nach dieser Phase ist ein Dienstplan fertig und alle Zellen sind endgültig belegt.

Wichtig: Früh- und Spätschichten werden nur in der dritten Phase belegt!

Für die Belegung des Plans in der zweiten Phase werden alle entsprechenden Rahmenbedingungen verwendet: Wünsche: rot (6.15), Wünsche schwarz (6.16), Wünsche: weiß (6.17), Belegung pro Schwester (6.22), Belegung pro Tag (6.25), globale Einschränkungen (6.28) und Muster (6.31).

6.45 Kosten-Fenster



Das Kosten-Fenster läßt sich im Menü: Optionen (6.12) ein- und ausschalten und zeigt für jede Schwester (6.5) die Kosten (6.45.1) an (siehe auch Optimierung eines Dienstplans (6.16.1)).

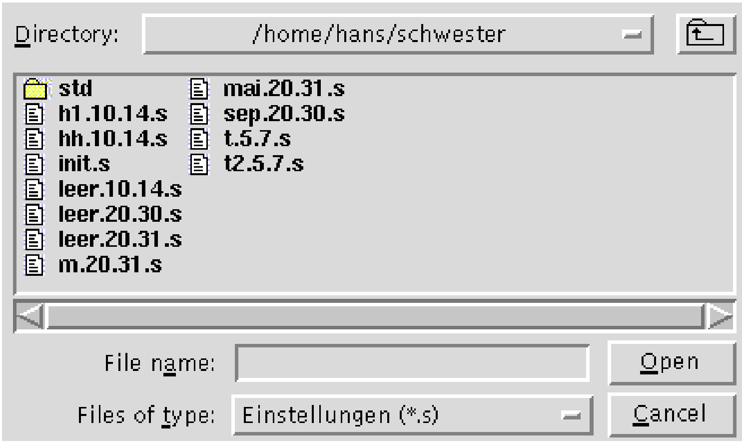
6.45.1 Kosten

Kosten werden zur Optimierung eines Dienstplans (6.16.1) benutzt. *Kosten* sind fiktiv, haben also keine Einheit oder Währung. Die Kosten eines gesamten Dienstplans entstehen für jede Schwester getrennt (Die Kosten können auch angezeigt werden: Kosten anzeigen (6.12)). Kosten entstehen entweder, wenn ein schwarzer oder weisser Wunsch verletzt wird (siehe auch Kosten für die Wünsche (6.18.1)) oder wenn zur Belegung eines Tages oder mehrerer Tage ein schlechtes Muster verwendet wird (s. Muster Phase 1 (6.29) , Muster Phase 2 (6.30) und Muster Phase 3 (6.31)). Der Benutzer kann an denselben Stellen die Werte für die Kosten verändern, die entsprechend erzeugt werden. Eine Veränderung beeinflusst die Gewichtung der weichen Bedingungen untereinander. So können z.B. weiche Wünsche (Wünsche: schwarz (6.16), Wünsche: weiß (6.17)) gegenüber der Verwendung von Mustern stärker gewichtet werden.

6.46 Wert ändern

Verschieden Werte werden mit diesem Dialog eingestellt. Der entsprechende Wert wird im umrandeten Feld eingegeben und durch Drücken der **RETURN-Taste** übernommen.

6.47 Einstellungen laden/speichern



Mit diesem Dialogfeld lassen sich Einstellungen (das sind die in INTERDIP veränderbaren Rahmenbedingungen) laden und speichern. In der Liste werden neben Dateien (Textsymbol) auch Verzeichnisse (Ordnersymbol) angezeigt. Die Dateien können geladen werden und in die Ordner kann gewechselt werden, jeweils mit Doppelklick auf den Namen. Zum Speichern gibt man im Feld *File Name* einen Namen an. Eine passende Endung wird automatisch erzeugt.

6.47.1 Einstellungsdateien

Nach dem Start von INTERDIP muss immer zuerst eine Datei geladen werden. Es gibt (in dieser Version) keine Möglichkeit, eine leere (das heisst ohne individuelle Einstellungen und Wünsche) neue anzulegen. Dafür muss zunächst eine leere bestehende Datei geladen sein.

Die Einstellungsdateien definieren neben den in INTERDIP veränderbaren Rahmenbedingungen auch die Größe eines Dienstplans, das ist die Anzahl der Zeilen und Spalten. Der Name einer Einstellungsdatei hat immer die Form: `irgendeinName`.*zeilen*.*spalten*`.s` Dabei ist *zeilen* die Anzahl der Zeilen und *spalten* die Anzahl der Spalten. Diese Größen können innerhalb eines INTERDIP-Laufs nicht verändert werden. Deshalb werden beim späteren Laden und Speichern in obigem Dialogfeld nur noch solche Dateien angezeigt, deren Name und damit deren Inhalt zur momentanen Größe passen.

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Arbeit war, unter Verwendung aktueller Techniken und mit dem gegebenen Constraint-Löser ein Werkzeug zu entwickeln, mit dem schnell und leicht gute Dienstpläne für Krankenstationen erstellt werden können. Außerdem sollten die verwendeten und entwickelten Konzepte möglichst auf andere, ähnliche Probleme übertragbar sein. Insbesondere sollte der Dienstplaner nicht nur für eine spezielle Krankenstation sondern für alle ähnlichen verwendbar und einfach zu erweitern sein.

Dieses Kapitel fasst die Arbeit zusammen und gibt einen Ausblick. Nach dem Vergleich mit verwandten Arbeiten folgen einige Bemerkungen zur Korrektheit und Vollständigkeit und zur Komplexität des dargestellten Verfahrens, dann die Zusammenfassung und am Schluss ein kurzer Überblick über den Verlauf des Projekts.

7.1 Verwandte Arbeiten

Unserer Arbeit verwandt sind vor allem solche, die das Dienstplanproblem für Krankenstationen unter Verwendung der Constraint-Technik behandeln. Alle beschriebenen Artikel legen, bis auf dort genannte Ausnahmen, der Dienstplanung im wesentlichen dieselben Rahmenbedingungen zugrunde, wie wir. Das unterstützt im übrigen die Annahme, dass es sich dabei um einen allgemeingültigen Rahmen handelt.

Das System SEPHOS [LA96] geht von einer anderen Organisation der Krankenstation als in unserem Fall aus. Dort können Schwestern offenbar keine Wünsche zur Dienstplanung äussern. Nur der Urlaub wird individuell berücksichtigt. Das unterscheidet den Rahmen doch wesentlich von unserem und vereinfacht das Problem wohl auch. Ansonsten benutzt SEPHOS die Constraint-Bibliothek ILOG und ist eine Impementieung in C. Anders als unser System kann SEPHOS die Anzahl der erforderlichen Schwestern minimieren.

HOROPLAN [DFM+94] implementiert einen interaktiven Dienstplaner als Teil eines mehrschrittigen Planungsverfahrens. Die Arbeit gibt Vorschläge, wie für die Schwestern angenehme Schichtfolgen erzeugt werden können. Die Modellierung scheint aber spezifischer zu sein, als unsere.

Diese beiden Arbeiten waren für uns wenig hilfreich. Die Beschreibung der dort entwickelten Verfahren ist zum einen zu knapp, um im Einzelnen nachvollzogen werden zu können. Zum anderen stand uns keine Testversion zur Verfügung, mit der wir hätten überprüfen können, wie mit den Verfahren gearbeitet werden kann, was für unseren interaktiven Ansatz ja sehr interessant gewesen wäre, und wie „gut“ die daraus resultierenden Dienstpläne sind, nach unserem Ermessen.

Die Arbeit [HW96] schlägt einen Dienstplaner vor, der einen Plan für typischerweise zwei Wochen generiert. Die Modellierung des Planungsproblems als CSP ist ähnlich der in Kapitel 3 beschriebenen. Die Autoren benutzen (wie bei SEPHOS) den ILOG-Constraint-Solver, allerdings als Lisp-Bibliothek.

Die für die Entstehung von INTERDIP wichtigste, unter anderem weil am besten dokumentierte Arbeit, ist [vdB93]. Der dort vorgeschlagene Dienstplaner generiert Pläne in zwei sehr unterschiedlichen Phasen, was nicht zuletzt uns zu der Formulierung des mehrphasigen Planers angeregt hat. Ein wesentlicher Unterschied des dort verfolgten Ansatzes zu unserer Arbeit ist die Benutzung von *work-patterns*: jeder Schwester ist eins von vier möglichen *work-patterns* zugeordnet. Ein Work-Pattern hat die Länge von ein oder zwei Wochen. Die Belegung des Plans geschieht dann streng anhand dieser Muster, wobei zunächst der komplette Zeitraum mit Mustern aufgefüllt wird und anschließend diese Muster um ein bis zwei Tage nach vorne oder hinten verschoben werden, um Belegungsgrenzen zu erfüllen. Die Muster von INTERDIP verallgemeinern den Ansatz der Work-Patterns. Im übrigen läßt das Verfahren von [vdB93] keine Planung der Nachtschichten zu, diese müssen vor der Planung feststehen.

Für zukünftige Verbesserungen von INTERDIP interessant ist auch das Projekt CONPLAN. Die Arbeiten hierzu [aHT95, aH96] beschäftigen sich mit weichen Constraints und deren Repräsentation durch „Preference Orderings“. In diesem Rahmen wird ein eigener Constraintlöser implementiert und damit ein Dienstplaner für Krankenstationen entworfen. Im Rahmen unserer Arbeit konnten diese Ergebnisse nicht berücksichtigt werden, da wir den vorliegenden Constraintlöser nicht verändern wollten (siehe auch Abschnitt 7.5).

7.2 Korrektheit und Vollständigkeit des Verfahrens

In der Informatik ist man immer an der Korrektheit und Vollständigkeit eines Verfahrens interessiert. Korrektheit in diesem Fall ist die Frage, ob mit dem Varfahren generierte Pläne korrekt sind. Bei der Vollständigkeit geht es um das Problem, ob, wenn es einen korrekten Plan gibt, dieser auch mit INTERDIP generiert werden kann. Nachdem INTERDIP ein interaktives und kein vollautomatisches Verfahren ist, können diese Fragen immer nur im Zusammenhang mit der Bedienung durch den Benuzter betrachtet werden.

INTERDIP generiert soweit korrekte Pläne, soweit der Benutzer die Rahmenbedingungen korrekt festlegt. So hat er unter anderem dafür zu sorgen, dass die Grenzen für die Arbeitszeiten jeder Schwester im Planungszeitraum richtig angegeben sind. Er muss dafür konkret angeben, wieviele Schichten jede Schwester arbeiten darf und muss. Man könnte sich natürlich vorstellen, dass das aus vertraglichen Daten und den Personaldaten der Schwestern über einen längeren Zeitraum automatisch geschieht. Das macht INTERDIP nicht. Auch kann der Benutzer selbständig Einschränkungen, die eigentlich gesetzliche harte Bedingungen darstellen, außer Kraft setzen. Werden alle diese Dinge korrekt spezifiziert, erzeugt INTERDIP nur korrekte Pläne. Hierfür ist kein formaler Beweis nötig. Nachdem die entsprechenden Bedingungen als harte Constraints behandelt werden, ist es Sache des Constraint-Lösers, dafür zu sorgen, dass keine dieser Bedingungen verletzt wird und damit alle generierten Pläne korrekt sind.

Allerdings sind nicht alle möglichen Bedingungen, die man an einen Dienstplan stellen könnte, mit INTERDIP auch formulierbar. Eine denkbare Bedingung wie „von den Schwester A, B und C muss jeden Tag mindestens eine in Früh- oder Spätschicht arbeiten“ oder ein komplexer Wunsch wie „Schwester Anna möchte / möchte nicht mit Schwester Berta in einer Schicht arbeiten“ kann mit der aktuellen Implementierung nicht formuliert werden. Wie der Wunsch behandelt werden könnte, ist in Abschnitt 5.5.2 beschrieben. Die genannte Bedingung ist technisch trivial in das bestehende System integrierbar. Was dazu vor allem fehlt, ist eine elegante allgemeine Beschreibung komplex zusammenhängender Constraints.

Auch die Vollständigkeit des Verfahrens wird nicht automatisch erfüllt. Für konkrete Planungsprobleme für einen Monat und 20 Schwestern wird tatsächlich oft nicht beim ersten Versuch (in angemessener Zeit) eine zufriedenstellende Lösung automatisch gefunden. Der Benutzer hat aber mit dem Debugger und der händischen Bearbeitung mit Konsistenz-Überprüfung die Möglichkeiten, solche Probleme mit relativ geringem Aufwand zu lokalisieren und zu lösen. Jedenfalls zeigten das unsere Tests. Insgesamt hängt die Vollständigkeit des automatischen Teils der Generierung an den beiden Eigenheiten unseres Verfahrens: der phasenweisen und der Belegung in Mustern.

Die Belegung in Mustern ist einerseits vollständig: da Dienstpläne nur nach den vorgegebenen Mustern belegt werden, sind die trivialen einelementigen Muster notwendig, um im Prinzip jeden möglichen Plan auch generieren zu können. Diese stehen standardmässig zur Verfügung, können aber vom Benutzer entfernt werden. Die Vollständigkeit kann aber andererseits nur im Zusammenhang mit dem zeitlichen Rahmen gesehen werden. Stehen für die Belegung einer Phase nämlich nur die trivialen Muster zur Verfügung, werden alle Zellen einzeln belegt, wodurch sich die Anzahl der tatsächlich untersuchten Belegungen wieder erhöht. Also sollten längere Muster den einelementigen vorangestellt werden. Insgesamt zeigt die Erfahrung, dass die Art und Reihenfolge der Muster eine grosse Rolle für die Vollständigkeit des Verfahrens spielt.

Die Frage der Vollständigkeit der Belegung in Phasen schließlich hat zwei widersprüchliche Aspekte. Einerseits kann in seltenen Fällen durch die konkrete Belegung in einer Phase in einer späteren keine Lösung gefunden werden, obwohl in der früheren Phase auch eine andere Belegung möglich gewesen wäre, die der späteren weniger Probleme gemacht hätte. Nachdem frühere Phasen nicht automatisch rückgängig gemacht werden (kein Backtracking über die Phasen stattfindet), hat INTERDIP selbst keine Möglichkeit, solche Probleme zu beheben. Andererseits aber scheint die Aufteilung der Belegung in Phasen die spezifischen Probleme, die für die Zuordnung der einzelnen Schichten existieren, recht gut zu lokalisieren und damit in aller Regel gut beheben zu können. Zum Beispiel müssen in der zweiten Phase nur die Probleme gelöst werden, die mit der Belegung der Nachtschichten zusammenhängen.

7.3 Komplexität

In dieser Arbeit wurden einige Angaben zur Komplexität des Verfahrens – die Anzahl der Möglichkeiten, die im schlechtesten Fall getestet werden müssen – gemacht. Dabei wurde davon ausgegangen, dass die einzelnen Phasen der Generierung unabhängig voneinander sind. Das ist so nicht richtig.

Tatsächlich muss der Benutzer (manchmal) eigenhändig phasenweise Belegungen rückgängig machen. Insofern findet ein Backtracking über die Phasen statt. Immer aber müssen höchstens „ein paar“ Phasen rückgängig gemacht werden, bis eine zufriedenstellende Lösung gefunden ist. Daraus eine verlässliche generelle Aussage über die Komplexität des Verfahrens zu machen, ist kaum möglich.

Insgesamt sollten die obigen Angaben als ein ungefährer Richtwert gelesen werden, nicht zuletzt, da diese Werte extrem gross sind, und ohnehin nicht alle Möglichkeiten getestet werden können.

7.4 Zusammenfassung

Mit INTERDIP ist es uns gelungen, innerhalb einiger Monate mit einem bestehenden Constraint-Solver einen interaktiven Dienstplaner für Krankenstationen zu entwickeln und zu implementieren.

Getestet wurde das Verfahren für konkrete Rahmenbedingungen einer Krankenstation eines münchener Krankenhauses für die Monate des vergangenen Jahres. Dabei konnten insbesondere mit der letzten Version des Dienstplaners, wie er hier beschrieben ist, korrekte und brauchbare bis gute Pläne in viel kürzerer Zeit als von Hand generiert werden.

Von Hand benötigt eine solche Planung einen Tag, mit INTERDIP die eigentliche Planung einige Minuten. Hinzu kommt noch die Erfassung der Rahmenbedingungen, und die Verwaltung der Personaldaten, die einen halben Tag in Anspruch nehmen können.

Das Ziel war nicht, einen vollautomatischen Dienstplaner zu entwickeln, sondern dem Benutzer ein Werkzeug an die Hand zu geben, mit dem er schnell und einfach gute Pläne erstellen kann. Das ist mit dieser Arbeit, bis auf die genannten Einschränkungen gelungen.

Auch ist INTERDIP recht allgemein gehalten und leicht erweiterbar. Ausgewählte Beispiele sind hier vorgeschlagen. Leicht erweiterbar ist das System vor allem, wegen des allgemeinen Ansatzes der Generierung in homogenen Phasen und der ausdruckstarken Muster und globalen Einschränkungen.

Insgesamt hat sich der interaktive Ansatz, der sich der besonderen Fähigkeiten des Menschen bedient, wieder einmal als tragfähig erwiesen.

7.5 Ausblick

Ein Teil der möglichen Erweiterung wurde schon genannt und dafür konkrete Möglichkeiten der Implementierung vorgeschlagen (Abschnitt 5.5).

Die wichtigste Erweiterung von INTERDIP ist die um komplexere Wünsche und Bedingungen. Dafür muss zunächst eine passende Moglichkeit der Spezifikation solcher Bedingung gefunden werden. Die Implementierung ist dann wohl sehr einfach.

Eine weitere vielversprechende Fortführung dieser Arbeit besteht in der Verwendung moderner Verfahren des Soft-Constraint-Solving. Dafür stehen Ergebnisse zur Verfügung, die auch im Zusammenhang der Dienstplanung für Krankenhäuser entstanden sind [aHT95, aH96]. Genauso könnte ein eigener Soft-Constraint-Löser unter Verwendung neuer Ansätze der Programmierung von Constraint-Solvern realisiert werden, zum Beispiel mit der Sprache CHR (Constraint-Handling-Rules) [Frü95]. Die Verwendung eines Solvers, der weiche Constraints behandeln kann, vereinfacht deren Formulierung wesentlich, sodass hier beschriebene umständliche und damit zum Teil ineffiziente Verfahren ersetzt werden könnten.

Für eine Vermarktung des Programmes INTERDIP und auch zur Vereinfachung der konkreten Planung, wäre eine Einbettung des Dienstplaners in ein umfangreicheres System, das auch die Verwaltung von Arbeitszeiten, Urlaub und sonstigen Personladaten übernimmt, sicher sehr wichtig. Damit würden dem Benutzer einige Arbeiten abgenommen, die er mit INTERDIP immer noch von Hand machen muss. Die Verwaltung eines längeren Zeitraums ließe auch eine Verbesserung der Gerechtigkeit des Verfahrens zu. Kommt eine Schwester in der Planung für einen Monat schlecht weg (hohe Kosten) könnte sie im nächsten Monat bevorzugt behandelt werden, zum Beispiel indem ihr die Kosten des Vormonats als Bonus gutgeschrieben wird. Dadurch würde die Belegung des nächsten Monats für diese Schwester von vorneherein weniger Kosten erzeugen und damit die Schwester eine bessere Verteilung der Schichten bzw. mehr Wünsche erfüllt bekommen.

7.6 Überblick über den Verlauf des Projekts

Diese Arbeit wurde von Oktober 1996 bis Mai 1997 von der Siemens Nixdorf Informationssysteme AG unterstützt und dort in dem Team von Klaus Dässler begonnen. Dieses Team gibt es heute leider nicht mehr. Fortgeführt und vollendet wurde es an der Ludwig Maximilian Universität am Lehrstuhl Bry.

Slim Abdennadher, der Betreuer an der LMU, und der Autor dieser Diplomarbeit haben eine Kurzform davon zum 12. Workshop logische Programmierung – WLP'97 eingereicht, der dort demnächst erscheint [AS97].

Anhang A

Programmtext

Im Anhang A ist der komplette Programmtext von INTERDIP gegeben.

A.1 interdip

Das Unix-Shell-Script `interdip` ruft Prolog so auf, dass die Datei `interdip.pro` geladen wird. Der evtl. angegebene Parameter gibt die Zeit in Dekunden an, nach der (eine Testversion von) INTERDIP automatisch beendet wird.

```
#!/bin/sh

# $1 ist jetzt die "time_til_kill" (siehe interdip.tcl)
tkprolog -c interdip.pro $1
```

A.2 H1.10.14.s

Folgende Datei ist eine Einstellungsdatei mit den Prädikaten der Spezifikation. Sie definiert ein Dienstplanproblem für 10 Schwestern und 14 Tage.

```
laden_und_speichern([laden_und_speichern / 1,schwestern / 1,tage / 1,
        zeitschranke / 1,erklaerungen / 1,sonntage / 1,muster / 3,
        belegung / 4,bereich/4,wuensche / 3,einschraenkung / 2]).
schwestern(10).
tage(14).
zeitschranke(10).
erklaerungen(1).
sonntage([1,7,8,14]).
muster(1,[0,'.','.','.','.','.',0],0).
muster(1,['.','.','.',0],5).
muster(1,['.','.'],10).
muster(1,['.'],15).
muster(1,[0],0).
muster(2,[3,3,0,0],0).
muster(2,['.'],0).
muster(2,[3],10).
muster(2,[0],0).
muster(3,[1,1,1],0).
muster(3,[2,2,2],0).
muster(3,[1],3).
muster(3,[2],3).
muster(3,[0],0).
muster(3,[3],0).
muster(3,['.'],0).
belegung(1,1,5,8).
belegung(1,2,5,8).
belegung(1,3,5,8).
belegung(1,4,5,8).
belegung(1,5,5,8).
belegung(1,6,5,8).
belegung(1,7,5,8).
belegung(1,8,5,8).
belegung(1,9,5,8).
belegung(1,10,5,8).
belegung(1,11,5,8).
belegung(1,12,5,8).
belegung(1,13,5,8).
belegung(1,14,5,8).
belegung(2,0,0,0).
belegung(2,1,0,0).
belegung(2,2,0,0).
belegung(2,3,0,0).
belegung(2,4,0,0).
belegung(2,5,1,1).
belegung(2,6,1,1).
belegung(2,7,1,2).
belegung(2,8,2,2).
belegung(2,9,0,0).
belegung(2,10,0,0).
belegung(3,0,0,0).
belegung(3,1,0,0).
belegung(3,2,0,0).
belegung(3,3,0,0).
belegung(3,4,2,2).
belegung(3,5,2,3).
belegung(3,6,3,3).
belegung(3,7,0,0).
belegung(3,8,0,0).
belegung(3,9,0,0).
belegung(3,10,0,0).
bereich(3,1,0,5).
bereich(3,2,0,5).
bereich(3,3,0,5).
bereich(3,4,0,5).
bereich(3,5,0,5).
bereich(3,6,0,5).
bereich(3,7,0,5).
```

```
bereich(3,8,0,5).
bereich(3,9,0,5).
bereich(3,10,0,5).
bereich(1,1,9,10).
bereich(1,2,9,10).
bereich(1,3,9,10).
bereich(1,4,9,10).
bereich(1,5,9,10).
bereich(1,6,9,10).
bereich(1,7,9,10).
bereich(1,8,9,10).
bereich(1,9,9,10).
bereich(1,10,9,10).
bereich(2,1,0,0).
bereich(2,2,0,5).
bereich(2,3,0,5).
bereich(2,4,0,5).
bereich(2,5,0,5).
bereich(2,6,0,5).
bereich(2,7,0,5).
bereich(2,8,0,5).
bereich(2,9,0,5).
bereich(2,10,0,5).
wuensche(rot,10000,[[0,0,2,2,'','','','','','',0,0,'',''],
        ['','','','','','','','','','','','','',''],
        ['','','','',0,0,'','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        [0,'','','','','',0,0,'','','','','',''],
        ['','','','','','',0,0,0,'','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        [0,1,1,'','','','',0,3,3,'','','',0]]).
wuensche(weiss,2,[[0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0],
        [0,'','','','','',0,0,'','','','','',0]]).
wuensche(schwarz,5,[['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','',''],
        ['','','','','','','','','','','','','','']]).
einschraenkung('Nie mehr als 10 Tage am Stück',aus).
einschraenkung('Nach zwei Nachtschichten mindestens zwei Freischichten',aus).
einschraenkung('Nach vier Nachtschichten mindestens drei Freischichten',aus).
einschraenkung('11-Stunden-Regel (Nachtschicht)',ein).
einschraenkung('11-Stunden-Regel (Spätschicht)',ein).
```

A.3 interdip.pro

`interdip.pro` enthält die wichtigsten Prädikate von INTERDIP, unter anderem das Hauptprogramm `program/0`.

```
% interdip.pro


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Initialisierung
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


/* Folgende Zeilen definieren den Inhalt der Einstellungs-Dateien (*.s) */

laden_und_speichern([laden_und_speichern/1,schwestern/1,tage/1,
        zeitschranke/1,erklaerungen/1,sonntage/1,muster/3,belegung/4,
        bereich/4,wuensche/3,einschraenkung/2]).

:-dynamic([laden_und_speichern/1,schwestern/1,tage/1,
        zeitschranke/1,erklaerungen/1,sonntage/1,muster/3,belegung/4,
        bereich/4,wuensche/3,einschraenkung/2]).

% die Listen muessen identisch ein, leider weiss ich keine andere
% Formulierung dafuer.
% Gibts wohl auch nicht (lt. Tapani). 30.4.97
```

```
/* fuer folgendes siehe inter.pro:laden/1 */
tage(_).
schwestern(_).


:- import(const_domain).
:- import(tk).

:- include('std/constr.pro').
:- include('std/ds.pro').

:- include('inter.pro').
:- include('phasen.pro').
:- include('einschraenkungen.pro').
:- include('wuensche.pro').

:- dynamic loesung/3,breakpoint/2.



/* Die beste Loeung einer Phase ist immer die erste der
Faktenbasis. Die beste Loesung aller Phasen (wenn Phase nicht gebunden
ist), ist die beste Loesung der letzten Phase!
*/

beste_loesung(Phase,Loesung,Kosten) :- loesung(Phase,Loesung,Kosten),!.



tag(T) :-
        tage(Tage),
        for(1,T,Tage).


schwester(Name) :-
        schwestern(Schwestern),
        for(1,Name,Schwestern).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialisierung
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Labeling
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


belegen(Phase,DS) :-
        dsSW31(DS,SW31),
        dsK(DS,Kosten),

        VorPhase is Phase-1,
        beste_loesung(VorPhase,Plan,_),

        % sollte der Schalter process_events_stop von einem frueheren Abbruch
        % noch gesetzt sein, wird er hier zurueckgesetzt, damit die Berechnung
        % nicht sofort abbricht.
        unset_global(process_events_stop),

        ( current_global(debugger) ->
            % Zeitschranke wird sehr hoch gesetzt, damit Debuggen nicht
            % unterbrochen wird. Die Einbindung unten in timebounded/2 ist
            % aber noetig damit der Benutzer die Berechnung abbrechen kann.
            ZeitS is 60*60*24*10,
            % Debugging ohne Optimierung!!!
            timebounded((
                retractall(debugpoint(_)),
                tcl_string(Plan,PlanS),
                tcl('anzeigen ##',[PlanS]),
                belegen_plan_d(Phase,1,SW31,Kosten,Plan,Plan_,[]),
                asserta(loesung(Phase,Plan_,Kosten)),
                tcl_string(Kosten,KostenS),
                tcl('kosten_anzeigen ##',[KostenS]),
                flush_output
            ),ZeitS)
        ;
            zeitschranke(ZeitS),
            tcl_string(Plan,PlanS),
            tcl('anzeigen ##',[PlanS]),
            timebounded(
                minimize_maximum((
```

```
                    belegen_plan(Phase,1,SW31,Kosten,Plan,Plan_),
                    asserta(loesung(Phase,Plan_,Kosten)),
                    tcl_string(Plan_,Plan_S),
                    tcl('anzeigen ##',[Plan_S]),
                    tcl_string(Kosten,KostenS),
                    tcl('kosten_anzeigen ##',[KostenS]),
                    flush_output
                ),Kosten),
            ZeitS)
        ).

/* Dei eigentlichen Praedikate zur Belegung. belegen_plan belegt den
ganzen Plan, belegen_zeile eine Zeile und belegen_block ein ganzes
Muster
*/
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ohne Debugging
%

/* belegen_plan(Phase,Schwester,SW31,Kosten,Plan,Plan_)
*/
belegen_plan(_,_,[],[],[],[]).

belegen_plan(Phase,Schw,[S|Ss],[K|Ks],[P|Ps],[P_|P_s]) :-
        belegen_zeile(Phase,Schw,1,S,K,P,P_,(
            Schw1 is Schw+1,
            belegen_plan(Phase,Schw1,Ss,Ks,Ps,P_s))).


belegen_zeile(_,_,_,[],K,[],[],Weiter) :- !,K=0,Weiter.

belegen_zeile(Phase,Schwester,Tag,Vars,Kosten,Plan,Plan_,Weiter) :-
        verfolgen(Schwester,Tag),
        Vars = [Var/_|_],
        Plan = [P|_],
        Muster = [P_|_],
        muster(Phase,Muster,K),
        phase_plan_map(Phase,P,Var,P_),
        belegen_block(
            Phase,
            Muster,
            Schwester,
            (Vars,Plan,Plan_,Tag),
            (VarsR,PlanR,Plan_R,Tag_),
            KostenWuensche ),
        Kosten_ ?>= 0,
        Kosten ?= K + KostenWuensche + Kosten_,
        belegen_zeile(Phase,Schwester,Tag_,VarsR,Kosten_,PlanR,Plan_R,Weiter).


/* belegen_block(Phase,Muster,Schwester,Listen,ListenR,KostenWuensche) :-
Listen = Vars,Plan,Plan_,Tag
ListenR = VarsR,PlanR,Plan_R,Tag_
*/
belegen_block(_,[],_,Listen,Listen,0) :- !.

belegen_block(Phase,[P_|Bs],Schwester,
 ([B/W|V_],[P|Ps],[P_|P_s],Tag),
 (VarsR,PlanR,Plan_R,Tag_),KostenWuensche) :-
        !,
        phase_plan_map(Phase,P,B,P_),
        process_events,
        kosten_wuensche(B,W,K),
        KostenWuensche_ ?>= 0,
        KostenWuensche ?= K + KostenWuensche_,
        Tag1 is Tag+1,
        belegen_block(Phase,Bs,Schwester,
            (V_,Ps,P_s,Tag1),
            (VarsR,PlanR,Plan_R,Tag_),KostenWuensche_).

belegen_block(_,_,_,([],[],[],_),([],[],[],_),0).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mit Debugging
%

/* belegen_plan_d(Phase,Schwester,SW31,Kosten,Plan,Plan_,DebugPoints)
*/
belegen_plan_d(_,_,[],[],[],[],DPs) :-
        process_events_pause(DPs).

belegen_plan_d(Phase,Schw,[S|Ss],[K|Ks],[P|Ps],[P_|P_s],DPs) :-
        belegen_zeile_d(Phase,Schw,1,S,K,P,P_,DPs,DPs_,(
            Schw1 is Schw+1,
            belegen_plan_d(Phase,Schw1,Ss,Ks,Ps,P_s,DPs_))).

/*belegen_zeile_d(Phase,Schwester,Tag,Vars,Kosten,Plan,Plan_,DPs,DPs_,Weiter):-
```

```
Weiter ist ein Goal, das nach vollständiger Erfüllung des Prädikats
auszuführen ist. Damit ist nach vollständiger Belegung des Plans der
Kontext mit allen Debugpunkten noch erhalten, sodass die Pause
eingeschaltet werden kann und später wieder zurückgesprungen werden
kann!!!
*/
belegen_zeile_d(_,_,_,[],K,[],[],DPs,DPs,Weiter) :- !,K=0,Weiter.

belegen_zeile_d(Phase,Schwester,Tag,Vars,Kosten,Plan,Plan_,DPs,DPs_,Weiter) :-
        Vars = [Var/_|_],
        Plan = [P|_],
        CatchMask = d(Schwester,Tag),
        context((
            process_events(DPs),
            Muster = [P_|_],
            muster(Phase,Muster,K),
            phase_plan_map(Phase,P,Var,P_),
            belegen_block_d(
                Phase,
                Muster,
                Schwester,
                (Vars,Plan,Plan_,Tag),
                (VarsR,PlanR,Plan_R,Tag_),
                KostenWuensche,
                [CatchMask|DPs]),
            % folgendes muss nach belegen_block_d/ stehen, damit evtl.
            % eingeschalteter Einzelschrittmodus auch nach Backtracking
            % erkannt wird.
            ( current_global(einzelschritt) ->
                throw(CatchMask)
            ; true ),
            Kosten_ ?>= 0,
            Kosten ?= K + KostenWuensche + Kosten_,
            ((breakpoint(Schwester,BreakTag),Tag=<BreakTag,BreakTag<Tag_) ->
                throw(CatchMask)
            ; true ),
            belegen_zeile_d(Phase,Schwester,Tag_,VarsR,Kosten_,PlanR,Plan_R,
                [CatchMask|DPs],DPs_,Weiter)
        ),[catch(CatchMask) => belegen_zeile_d_auswahl(Phase,Schwester,Tag,
                                    Vars,Kosten,Plan,Plan_,DPs,DPs_,Weiter)]).

belegen_zeile_d_auswahl(Phase,Schwester,Tag,Vars,Kosten,Plan,Plan_,
 DPs,DPs_,Weiter) :-
        Vars = [Var/_|_],
        Plan = [P|_],
        CatchMask = d(Schwester,Tag),
        context((
            process_events(DPs),
            findall([Muster1,K1,Muster1],(
                Muster1 = [P_|_],
                muster(Phase,Muster1,K1),
                phase_plan_map(Phase,P,Var,P_),
                belegen_block(
                    Phase,
                    Muster1,
                    Schwester,
                    (Vars,Plan,_,Tag),
                    (_,_,_,_),
                    _)
            ),MKMs),
            d_auswahl(Schwester,Tag,MKMs,Muster,K),
            belegen_block_d(
                Phase,
                Muster,
                Schwester,
                (Vars,Plan,Plan_,Tag),
                (VarsR,PlanR,Plan_R,Tag_),
                KostenWuensche,
                [CatchMask|DPs]),
            Kosten_ ?>= 0,
            Kosten ?= K + KostenWuensche + Kosten_,
            belegen_zeile_d(Phase,Schwester,Tag_,VarsR,Kosten_,PlanR,Plan_R,
                        [CatchMask|DPs],DPs_,Weiter)
        ),[catch(CatchMask) => belegen_zeile_d_auswahl(Phase,Schwester,Tag,
                                    Vars,Kosten,Plan,Plan_,DPs,DPs_,Weiter)]).



/* belegen_block_d(Phase,Muster,Schwester,Listen,ListenR,KostenWuensche) :-
Listen = Vars,Plan,Plan_,Tag
ListenR = VarsR,PlanR,Plan_R,Tag_
*/
belegen_block_d(_,[],_,Listen,Listen,0,_) :- !.

belegen_block_d(Phase,[P_|Bs],Schwester,
 ([B/W|V_],[P|Ps],[P_|P_s],Tag),
 (VarsR,PlanR,Plan_R,Tag_),KostenWuensche,DebugPoints) :-
```

```
        !,
        phase_plan_map(Phase,P,B,P_),
        schnappen(Phase,Schwester,Tag,P,P_),
        process_events,
        kosten_wuensche(B,W,K),
        KostenWuensche_ ?>= 0,
        KostenWuensche ?= K + KostenWuensche_,
        Tag1 is Tag+1,
        belegen_block_d(Phase,Bs,Schwester,
            (V_,Ps,P_s,Tag1),
            (VarsR,PlanR,Plan_R,Tag_),KostenWuensche_,DebugPoints).

belegen_block_d(_,_,_,([],[],[],_),([],[],[],_),0,_).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Labeling
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Phasen - Plan
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% phase_plan_map(Phase,Plan,Var,Plan_) :-
phase_plan_map(1,.,0,0).
phase_plan_map(1,.,1,.).
phase_plan_map(2,.,0,.).
phase_plan_map(2,.,1,3).
phase_plan_map(2,0,0,0).
phase_plan_map(3,.,0,2).
phase_plan_map(3,.,1,1).
phase_plan_map(3,0,0,0).
phase_plan_map(3,3,0,3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Phasen - Plan
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Ausgabe eines Dienstplans
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

dr_plan(Phase) :-
        beste_loesung(Phase,P,_),
        rec_doall(P,Z,(
            rec_doall(Z,E,write_formatted('%2t',[E])),
            nl
        )),
        nl.

dr_plan :-
        dr_plan(_).


% LaTeX-Darstellung:
dr_plan_l :-
        nl,nl,
        beste_loesung(_Phase,Plan,_),
        write('\
    \\begin{tabular}{r|r*{'),
        tage(Tage),
        Tage1 is Tage-1,
        write(Tage1),
        write('}{c}|}\n'),
        write('&Tag'),
        doall((for(1,_,Tage1),write('& '))),
        write('\\\\'),
        nl,
        write('Schwester'),
        doall((for(1,I,Tage),write_formatted('&%02d',[I]))),
        write('\\\\ \\hline'),
        rec_doall_1(Plan,Z,Schwester,(
            nl,
            write(Schwester),
            rec_doall(Z,E,write_formatted('&%2t',[E])),
            write('\\\\')
        )),
        write('\
 \\hline\n\
```

```
    \\end{tabular}\n'),
        nl,nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ausgabe eines Dienstplans
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Hauptprogramm
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

program :-
        ( user_parameters([TimeTilKill]) ->
            true
        ;
            TimeTilKill = 0
        ),
        set_global(time_til_kill,TimeTilKill),
        % TimeTilKill ist, wenn ungleich 0, die Zeit in Sekunden,
        % nach der interdip automatisch beendet wird. Das ist für
        % die im Internet angebotene Demoversion.

        tk([file('interdip.tcl')]),

        s erz_ds(DS),

        repeat,
        tk_do_one_event(0),
        (current_global(phase) ->
            get_global(phase,Phase),
            phase s Phase s not phase(Phase,DS),
            unset_global(phase)
        ;
            unset_global(phase)
        ),
        (current_global(konsistenz_pruefen) ->
            get_global(konsistenz_pruefen,Plan),
            konsistenz_pruefen(Plan,DS),
            unset_global(konsistenz_pruefen)
        ;
            unset_global(konsistenz_pruefen)
        ),
        fail.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Hauptprogramm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.4 phasen.pro

Diese Datei definiert die Prädikate zur Berechnung einer Phase. Die eigentliche Belegung eines Plans befindet sich aber in `interdip.pro`.

```
% phasen.pro


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Phase
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

phase(0,_) :-
        findall(L,(schwester(_),findall('.',tag(_),L)),NullteLoesung),
        findall(0,schwester(_),NullteLoesungKosten),
        asserta(loesung(0,NullteLoesung,NullteLoesungKosten)),
        !,
        fail.


phase(Phase,DS) :-
        Phase > 0,
        VorPhase is Phase-1,
```

```
        % alte Ergebnisse loeschen
        doall((
            loesung(Ph,_,_),
            Ph >= Phase,
            retract(loesung(Ph,_,_))
        )),

        % Existiert zu der dieser Phase vorausgehenden Phase keine Loesung,
        % dann wird sie berechnet.
        ( loesung(VorPhase,_,_) ->
            true
        ;
            phase s VorPhase s not phase(VorPhase,DS) ),
        beste_loesung(VorPhase,Plan,_),

        constraints_eintragen(Phase,Plan,DS),

        doall(belegen(Phase,DS)),

/* Damit die Belegung der Var. wieder aufgehoben wird (die gefundenen
Loesungen sind ja in loesung(Phase,L,_) gespeichert), damit die Var. in der
naechsten Phase wieder benutzt werden koennen und DS nicht neu erzeugt
werden muss, wird hier abgeschnitten und gescheitert.
*/
        !,fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Phase
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Constraints eintragen
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

constraints_eintragen(Phase,Plan,DS) :-
        dsS0(DS,S0),
        S0 in [0,1],

        dsS31(DS,S31),
        s plan_eintragen(Plan,S31),
        dsS32(DS,S32),
        transpose(Plan,PlanT), % zur Ermittlung der Belegungsgrenzen
        s belegung_eintragen(Phase,PlanT,S32),

        s einschraenkungen_eintragen(Phase,Plan,S31),

        s wuensche_eintragen(Phase,10000,DS),

        dsAT(DS,AusgleichTage),
        s bereich_eintragen(Phase,AusgleichTage,S31).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Constraints eintragen
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Plan eintragen
%
% Hier wird der bereits berechnete Plan (aus vorangegangenen Phasen) in
% die Variablen dieser Phase eingetragen. Dadurch koennen einige
% Var. instantiiert werden.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


/* plan_eintragen(Plan,S31) traegt den bish. Plan in die Var. ein: die Var.
        der neuen Phase werden mit den Werten belegt, die sich aus dem
        bisherigen Plan ergeben.
*/
plan_eintragen(Plan,S31) :-
        mapper([Plan,S31],[PlanZ,S31Z],
            mapper([PlanZ,S31Z],[Bel,S],(Bel \= '.' -> S = 0 ;true))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plan eintragen
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Einschraenkungen eintragen
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% einschraenkungen_eintragen(Phase,Plan,S31) :-

einschraenkungen_eintragen(Phase,Plan,S31) :-
        einschraenkungen_eintragen(Phase,Plan,S31,1).

einschraenkungen_eintragen(_,[],[],_).
einschraenkungen_eintragen(Phase,[P|Ps],[S|Ss],Schwester) :-
        einschraenkungen_eintragen(Phase,P,S,Schwester,1),
        !,
        Schwester1 is Schwester+1,
        einschraenkungen_eintragen(Phase,Ps,Ss,Schwester1).

einschraenkungen_eintragen(_,[],[],_,_).
einschraenkungen_eintragen(Phase,Ps,Ss,Schwester,Tag) :-
        einschraenkung_def(Phase,Name,Ps,Ss,E),
        einschraenkung(Name,ein),
        !,
        (einschraenkung,Name,Phase,Schwester,Tag) c E,
        [_|PsR] = Ps,
        [_|SsR] = Ss,
        Tag1 is Tag+1,
        einschraenkungen_eintragen(Phase,PsR,SsR,Schwester,Tag1).
einschraenkungen_eintragen(Phase,[_|Ps],[_|Ss],Schwester,Tag) :-
        Tag1 is Tag+1,
        einschraenkungen_eintragen(Phase,Ps,Ss,Schwester,Tag1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Einschraenkungen eintragen
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Belegung eintragen
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% belegung_eintragen(Phase,PlanT,S32) :-

belegung_eintragen(Phase,PlanT,S32) :-
        mapper1([S32,PlanT],[S,PT],I,(
            const_sum(S,SS),
            % fuer die erste Phase hat belegung/4 ja eine andere Bedeutung...
            ( Phase == 1 ->
                belegung(Phase,I,Min,Max)
            ;
                count(member('.',PT),SM),
                (I,'Belegungsverteilung') c belegung(Phase,SM,Min,Max)
            ),
            (belegung,Phase,I) c SS ?>= Min,
            (belegung,Phase,I) c SS ?=< Max
        )).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Belegung eintragen
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Bereich eintragen
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% bereich_eintragen(Phase,AusgleichTage,S31) :-

bereich_eintragen(Phase,AusgleichTage,S31) :-
        mapper1([S31,AusgleichTage],[S,AT],I,(
            bereich(Phase,I,Min,Max),
            list_sum(S,SS),
            (bereich,Phase,I) c SS+AT ?>= Min,
            (bereich,Phase,I) c SS+AT ?=< Max
        )).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bereich eintragen
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.5 ds.pro

Definiert die in Abschnitt 5.3.2 beschriebenen Datenstrukturen.

```
/* ds.pro

   Definiert die Datenstrukturen:


   [V1,V2,...]                                             s0

   [[1,1,V1],[1,2,V2],...]             ([Zeile,Spalte,V])      s1

   [1-[1-[V1],2-[V2],...],...]         ([Zeile-[Spalte-[V]]])  s21

   [1-[1-[V1],2-[V10],...],...]        ([Spalte-[Zeile-[V]]])  s22

   [[V1,V2,...],...]                   (Zeilenlisten)          s31

   [[V1,V10,...],...]                  (Spaltenlisten)         s32

jetzt mit Platz fuer die Wuensche: sw0,sw1,sw21,sw22,sw31,sw32

*/



% erz_vars(Vars) :- Variablen erzeugen

erz_vars(Vs) :-
        findall([S,T,_],(schwester(S),tag(T)),Vs).


% zeile_spalte_vertauschen(Vs,Vs_) :-

zeile_spalte_vertauschen([],[]).
zeile_spalte_vertauschen([[Z,S,V]|R],[[S,Z,V]|R_]) :-
        zeile_spalte_vertauschen(R,R_).
```

```
% Zugriff auf die Elemente der Datenstruktur:
/* folgende Praedikate werden jetzt von ds_zugriffe/2 generiert:
dsS0( ( S0,_,_,_,_,_, _,_,_,_,_, _,_,_,_,_, _),S0).
dsS1( (_, S1,_,_,_,_, _,_,_,_,_, _,_,_,_,_, _),S1).
dsS21((_,_,S21,_,_,_, _,_,_,_,_, _,_,_,_,_, _),S21).
...
*/

ds_zugriffe(Liste,DS) :-
        list_length(Liste,Stellen),
        functor(DS,',',Stellen),
        ds_zugriffe1(Liste,1,DS).

ds_zugriffe1([],_,_).
ds_zugriffe1([(Name,Var)|R],Stelle,S) :-
        arg(Stelle,S,Var),
        concat_atom([ds,Name],F),
        P =.. [F,S,Var],
        asserta(P),
        Stelle1 is Stelle+1,
        ds_zugriffe1(R,Stelle1,S).



/* ds_varstrukts(Namen,ZugriffListe) :- erzeugt fuer die Namen eine Liste fuer
              ds_zugriffe/2 und die Praedikate dsName_add/2 und dsName_sum/2.
              Das alles fuer "variable Datenstrukturen". Also z.B.:

ds_varstrukts(['K'],[('K',L),('K_list',(L,_))]) :-
        asserta(
dsK_add(DSA,A) :-
        dsK_list(DSA,(_,LA)),
        ds_var_add(LA,A) ),
        asserta(
dsK_sum(DSS) :-
        dsK_list(DSS,(S,LS)),
        ds_var_sum(LS,S) ).
*/

/* Ein Varstrukt ist eine Struktur, der nach der Erzeugung, also
gewissermassen zur Laufzeit, noch andere Strukturen hinzugefuegt
werden koennen (add) die dann geschlossen werden kann (sum). Oder so
aehnlich...
```

```
*/

ds_var_add(L,A) :-
        ( var(L) -> L = [A|_] ; L=[_|R],ds_var_add(R,A) ).

/* z.B. ds_var_add(L,a),ds_var_add(L,b),ds_var_add(L,c),ds_var_add(L,d). */

ds_var_sum(L,S) :-
        ds_var_sum_schliessen(L),
        list_sum_list(L,S).

ds_var_sum_schliessen([]) :- !.
ds_var_sum_schliessen([_|R]) :- ds_var_sum_schliessen(R).


ds_varstrukts([],[]).
ds_varstrukts([Name|RN],[(Name,L),(N2,(L,_))|RV]) :-
        concat_atom([Name,'_list'],N2),
        concat_atom([ds,Name,'_add'],Nadd),
        concat_atom([ds,Name,'_sum'],Nsum),
        concat_atom([ds,N2],Nlist),
        Hadd =.. [Nadd,HaddDS,HaddL],
        B1add =.. [Nlist,HaddDS,(_,BaddL)],
        B2add = ds_var_add(BaddL,HaddL),
        asserta((Hadd :- B1add,B2add)),
        Hsum =.. [Nsum,HsumDS],
        B1sum =.. [Nlist,HsumDS,(BsumS,BsumL)],
        B2sum = ds_var_sum(BsumL,BsumS),
        asserta((Hsum :- B1sum,B2sum)),
        ds_varstrukts(RN,RV).



/* erz_ds(Vars,DS) :- Datenstrukturen (s.o) erzeugt. Zusaetze sind
zusaetzliche mit obigen Praedikaten erzeugte variable
Strukturen. (INTERDIP benutzt das aktuell nicht!).
*/

erz_ds(Zusaetze,DS) :-

        ds_zugriffe([
        ('S0',S0),('S1',S1),('S21',S21),('S22',S22),('S31',S31),('S32',S32),
        ('SW0',SW0),('SW1',SW1),('SW21',SW21),('SW22',SW22),('SW31',SW31),
        ('SW32',SW32),
        ('W0',W0),('W1',W1),('W21',W21),('W22',W22),('W31',W31),('W32',W32),
        ('AT',AusgleichTage),
        ('K',Kosten)
    |Zusaetze],DS),

        erz_vars(Vs),

        S1 = Vs,
        var_tupel_list(Vs,S21),
        zeile_spalte_vertauschen(Vs,Vs_),
        var_tupel_list(Vs_,S22),
        var_list_einfach(S21,S31_),
        map(S31_,append,S31),
        append(S31,S0),
        var_list_einfach(S22,S32_),
        map(S32_,append,S32),
        system:copy_term(
            (S0,S1,S21,S22,S31,S32),
            (SW0,SW1,SW21,SW22,SW31,SW32)),
        map(S0,ylppa(P2 = P1/_,[P1,P2]),SW0),
        system:copy_term((S0,S1,S21,S22,S31,S32),(W0,W1,W21,W22,W31,W32)),
        map(SW0,ylppa(P1 = _/P2,[P1,P2]),W0),

        AusgleichTage = _,    % wird spaeter (implizit) erzeugt
        findall(_,schwester(_),Kosten).



erz_ds(DS) :-
        erz_ds([],DS).
```

A.6 wuensche.pro

`wuensche.pro` enthält die für die Behandlung der Wünsche nötigen Prädikate.

```
/* wuensche.pro
*/
```

```
/* wunsch(Phase,Wunsch,Var,AusgleichTage) :-
*/

% Erste Phase: Frei/Dienst-Schichten
% Belegt werden "Dienstschichten", d.h. eine 1 steht fuer "Dienst",
% 0 fuer "Frei"
wunsch(1,'',_,0).
wunsch(1,0,0,0).
wunsch(1,1,1,0).
wunsch(1,2,1,0).
wunsch(1,3,1,0).
wunsch(1,u,0,1).
wunsch(1,d,1,0).

% Zweite Phase: Nachtschichten
wunsch(2,'',_,0).
wunsch(2,0,0,0).
wunsch(2,1,0,0).
wunsch(2,2,0,0).
wunsch(2,3,1,0).
wunsch(2,u,0,0).
wunsch(2,d,0,0).

% Dritte Phase: Fruehschichten
wunsch(3,'',_,0).
wunsch(3,0,0,0).
wunsch(3,1,1,0).
wunsch(3,2,0,0).
wunsch(3,3,0,0).
wunsch(3,u,0,0).
wunsch(3,d,_,0).



/* wuensche_setzen(Farbe,KostenDafuer,Wuensche) :-
        die in die Maske eingegebenen
        Wuensche werden in Datenbasis uebernommen
*/
wuensche_setzen(Farbe,KostenDafuer,Wuensche) :-
        retract(wuensche(Farbe,_,_)),
        asserta(wuensche(Farbe,KostenDafuer,Wuensche)).
```

```
/* wuensche_eintragen(Phase,Schwelle,DS) :-
Die Wuensche werden fuer Phase eingetragen. Schwelle ist der Kostenwert,
ab dem Wuensche als harte Constraints behandelt werden. z.B.
wuensche_eintragen(1,10000,DS).
*/

wuensche_eintragen(Phase,Schwelle,DS) :-
        dsAT(DS,AusgleichTage),
        findall(KostenDafuer/Farbe,wuensche(Farbe,KostenDafuer,_),W),
        sort(W,W_),
        reverse(W_,Wuensche),
        wuensche_eintragen(Phase,Wuensche,Schwelle,AusgleichTage,DS).

wuensche_eintragen(_,[],_,[],_) :- !.
wuensche_eintragen(Phase,[KostenDafuer/Farbe|R],Schwelle,AusgleichTage,DS) :-
        ( KostenDafuer >= Schwelle ->
            wuensche_eintragen_hart(Phase,Farbe,AusgleichTage1,DS)
        ;
            wuensche_eintragen_weich(Phase,Farbe,AusgleichTage1,DS) ),
        wuensche_eintragen(Phase,R,Schwelle,AusgleichTage2,DS),
        wuensche_eintragen_ausgleich_summieren(AusgleichTage1,AusgleichTage2,
            AusgleichTage).


/* wuensche_eintragen_ausgleich_summieren(AusgleichTage1,AusgleichTage2,
AusgleichTage) :-
*/
wuensche_eintragen_ausgleich_summieren(A,[],A) :- !.
wuensche_eintragen_ausgleich_summieren([A1|R1],[A2|R2],[A|R]) :-
        A ?= A1 + A2,
        wuensche_eintragen_ausgleich_summieren(R1,R2,R).


% wuensche_eintragen_hart(Phase,Farbe,AusgleichTage,DS)
/* Hier wird AusgleichTage als Liste erzeugt! */
wuensche_eintragen_hart(Phase,Farbe,AusgleichTage,DS) :-
        dsS31(DS,S31),
        wuensche(Farbe,_KostenDafuer,Ws),
        mapper1([Ws,S31,AusgleichTage],[W,S,AT],NrSchwester,(
            mapper1([W,S,AT_],[We,Se,AT_e],NrTag,(
                (NrSchwester,NrTag) c wunsch(Phase,We,Se,AT_e)
            )),
            list_sum(AT_,ATS),
            AT ?= ATS )).
```

```
% wuensche_eintragen_weich(Phase,Farbe,AusgleichTage,DS)
/* Hier wird AusgleichTage als Liste erzeugt! */
wuensche_eintragen_weich(Phase,Farbe,AusgleichTage,DS) :-
        dsW31(DS,W31),
        wuensche(Farbe,KostenDafuer,Ws),
        mapper([Ws,W31,AusgleichTage],[W,S,O],(
            map(W,wuensche_eintragen_weich1(Phase,KostenDafuer),S)
        )).

wuensche_eintragen_weich1(Phase,KostenDafuer,W,WL) :-
        var(WL),
        !,
        wunsch(Phase,W,V,_),
        ( is_domain(V) ->        % s. die erste Variante von wunsch/3
            WL = [V/KostenDafuer|_]
        ;
            true
        ).
wuensche_eintragen_weich1(Phase,KostenDafuer,W,[_|R]) :-
        wuensche_eintragen_weich1(Phase,KostenDafuer,W,R).



% kosten_wuensche(Belegung,WunschListe,Kosten) :-

kosten_wuensche(B,[WK|R],Kosten) :-
        not var(WK),
        !,
        WK=W/K,
        ( B=W ->        % der Trigger !
                Kosten = Kosten_
        ;
                Kosten ?= K + Kosten_ ),
        kosten_wuensche(B,R,Kosten_).
kosten_wuensche(_,_,K) :- !,K=0.
```

A.7 einschraenkungen.pro

Diese Datei definiert die globalen Einschränkungen.

```
% einschraenkungen.pro

/* Hier werden die phasenspez. Einschraenkungen definiert
*/

% einschraenkung_def(Phase,Name,MusterPlan,MusterVar,Constraint).


einschraenkung_def(1,
        'Nie mehr als 10 Tage am Stück',
        /* die Summe ueber elf Tage <= 10 */
        _,
        [A,B,C,D,E,F,G,H,I,J,K|_],
        (A+B+C+D+E+F+G+H+I+J+K ?=< 10)).

einschraenkung_def(2,
        'Nach zwei Nachtschichten mindestens zwei Freischichten',
        ['.','.',0,'.'|_],
        [A|_],
        A ?= 0).

einschraenkung_def(2,
        'Nach vier Nachtschichten mindestens drei Freischichten',
        ['.','.','.','.',0,_,'.'|_],
        [A,B,C,D|_],
        (A ?= 0 , B ?=< C , C ?=< D)).

einschraenkung_def(2,
        '11-Stunden-Regel (Nachtschicht)',
        ['.','.'|_],
        [A,B|_],
        A ?=< B).

einschraenkung_def(3,
        '11-Stunden-Regel (Spätschicht)',
        ['.','.'|_],
        [A,B|_],
        A ?>= B).
```

A.8 inter.pro

`inter.pro` definiert die Prädikate für die interaktive Benutzerschnittstelle. Diese selbst wird durch `interdip.tcl` implementiert.

```
% inter.pro

/* Die Praedikate fuer die interaktive Benutzerschnittstelle
*/


/* Folgendes dient der Anzeige der Statuszeile. Das muss hier stehen, weil
es in anderen Dateien benutzt wird und deshalb vorher als Operator
deklariert sein muss.
*/
:- op(999,fy,s).
:- op(998,xfy,s).
s(Goal) :-
        Goal =.. [Name|_],
        ( tcl('statuszeile "##"',[Name]),tk_loop_while_events
                ;
          tcl('statuszeile_ruecksetzen'),tk_loop_while_events,fail ),
        Goal,
        ( tcl('statuszeile_ruecksetzen'),tk_loop_while_events
                ;
          tcl('statuszeile "##"',[Name]),tk_loop_while_events,fail ),
        true.
s(Name,Goal) :-
        ( tcl('statuszeile "##"',[Name]),tk_loop_while_events
                ;
          tcl('statuszeile_ruecksetzen'),tk_loop_while_events,fail ),
        Goal,
        ( tcl('statuszeile_ruecksetzen'),tk_loop_while_events
                ;
          tcl('statuszeile "##"',[Name]),tk_loop_while_events,fail ),
        true.


/* Zum Protokollieren fehlgeschlagener Constraints
*/
:- op(999,fy,c).
:- op(998,xfy,c).
c(C) :- C,! ; inkonsistent(C),!,fail .
c(N,C) :- C,! ; inkonsistent(N,C),!,fail .
```

```
/* Festlegen der Bereiche aus der Benutzeroberflaeche
*/
bereich_setzen(Phase,Liste) :-
        retractall(bereich(Phase,_,_,_)),
        rec_doall_1(Liste,[Min,Max],I,assertz(bereich(Phase,I,Min,Max))).

bereich_holen(Phase,Liste) :-
        findall([Min,Max],bereich(Phase,_,Min,Max),Liste).



einschraenkungen_holen(L) :-
        findall([EinAus,Name,Phase],(
            einschraenkung_def(Phase,Name,_,_,_),
            ( einschraenkung(Name,ein) -> EinAus = ein ; EinAus = aus )
        ),L).

einschraenkungen_setzen(L) :-
        findall(Name,einschraenkung_def(_,Name,_,_,_),NL),
        retractall(einschraenkung(_,_)),
        mapper([L,NL],[EinAus,Name],(
            assertz(einschraenkung(Name,EinAus))
        )).



/* Festlegen der Belegungen aus der Benutzeroberflaeche
*/
belegung_setzen(Phase,Liste) :-
        rec_doall(Liste,[Rest,Min,Max],(
            retractall(belegung(Phase,Rest,_,_)),
            assertz(belegung(Phase,Rest,Min,Max))
        )).

belegung_holen(Phase,L) :-
        findall([TagRest,Min,Max],belegung(Phase,TagRest,Min,Max),L).



/* Festlegen der Muster aus der Benutzeroberflaeche
```

```
*/
muster_setzen(Phase,Liste) :-
        retractall(muster(Phase,_,_)),
        rec_doall(Liste,M,(append(B,[K],M),assertz(muster(Phase,B,K)))).

muster_holen(Phase,Liste) :-
        findall([M,K],muster(Phase,M,K),Liste).



/* wird von interdip.tcl veraendert!!!*/
:- dynamic verfolgen/2.


% zeigt die Position der Belegung im Dienstplan an

verfolgen(_,_) :- !. % d.h. die Verfolgung ist anfangs ausgeschaltet
verfolgen(Schwester,Tag) :-
        (
            tcl('verfolgen ## ## white',[Schwester,Tag]),
            tk_loop_while_events
        ;
            tcl('verfolgen ## ##',[Schwester,Tag]),
            tk_loop_while_events,fail
        ).


% zeigt den Stand der Belegung (mit Eintrag) im Dienstplan an

schnappen(_Phase,Schwester,Tag,_P,P_) :-
        (
            tcl('schnappen ## ## ##',[Schwester,Tag,P_]),
            tk_loop_while_events
        ;
            tcl('schnappen ## ##',[Schwester,Tag]),
            tk_loop_while_events,fail
        ).


% ermöglicht Tcl/Tk anstehende Ereignisse zu verarbeiten. Nichtblockierend.

process_events :-
        tk_loop_while_events,
        process_stop,
        !.


% mit debugging:
process_events(DebugPoints) :-
        tk_loop_while_events,
        (current_global(pause) -> process_events_pause(DebugPoints) ; true),
        process_stop,
        zurueck(DebugPoints),
        !.

% mit debugging, nur fuer Pausenbehandlung (aehnlich process_events/1,
% aber die Tk-Events werden blockierend abgefragt!):
process_events_pause(DebugPoints) :-
        pause_ein,
        repeat,
        tk_do_one_event(0),
        process_stop,
        zurueck(DebugPoints),
        not current_global(pause),
        !.

/* Falls die globale Var. process_events_stop gesetzt is, wird die
Berechnung abgebrochen
*/
process_stop :-
        current_global(process_events_stop),
        !,
        unset_global(process_events_stop),
        throw(timeout).
process_stop.

/* Falls die globale Var. zurueck gesetzt ist, wird an die angegebene
Stelle zurueckgesprungen (Belgung rueckgaengig gemacht).
*/
zurueck(DebugPoints) :-
        current_global(zurueck),
        !,
        get_global(zurueck,(Schwester,Tag)),
        unset_global(zurueck),
        member(d(S,T),DebugPoints),
        T =< Tag,
        S =< Schwester,
        tcl('verfolgen ## ## white',[S,T]),
```

```
        tk_loop_while_events,
        throw(d(S,T)).
zurueck(_).

/* Leitet den Abbruch der Berechnung ein.
*/
process_events_stop :-
        set_global(process_events_stop,nn).


/* loeschen :- alle Loesungen werden geloescht */
loeschen :-
        retractall(loesung(_,_,_)).



/* Praedikate zum Laden und Speichern von Einstellungen/Daten

Wenn die geladene Date nicht mit den aktuellen Einstellungen
kompatibel ist (dieselbe Liste laden_und_speichern(L) und dieselben
Fakten schwestern(S) -- Anzahl der Zeilen -- und tage(T) -- Anzahl der
Spalten --), wird das Programm mit Fehler beendet.
*/
laden(DatNam):-
        consult(DatNam),
        findall(T,tage(T),[T,T]),
        findall(S,schwestern(S),[S,S]),
        findall(L,laden_und_speichern(L),[L,L]),
        reconsult(DatNam),
        !.
laden(_) :-
        write(user_error,'\
Programmabbruch wegen undefiniertem Zustand.\n\
Moegliche Ursachen:\n\
- Plangroesse der geladenen Datei ist ungleich der bisherigen\n\
- die geladene Datei stammt von einer aelteren Version\n\
- die Datei konnte aus einem anderen Grund nicht erfolgreich geladen werden\n\
'),
        halt.


speichern(DatNam) :-
        open(DatNam,write,Stream),
        laden_und_speichern(L),
```

```
        doall((
            member(F/A,L),
            functor(S,F,A),
            doall((
                S,
                writeq(Stream,S),
                write(Stream,'.'),
                nl(Stream)
            ))
        )),
        close(Stream).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Debugger
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- dynamic debugpoint/1.

debug_ein :-
        not current_global(phase),
        not current_global(debugger),
        set_global(debugger,nn).
debug_aus :-
        not current_global(phase),
        current_global(debugger),
        unset_global(debugger).


pause_ein :-
        set_global(pause,nn),
        tcl('global pause ; set pause 1'),
        tk_loop_while_events.
pause_aus :-
        unset_global(pause),
        tcl('global pause ; set pause 0'),
        tk_loop_while_events.


einzelschritt_ein :-
        set_global(einzelschritt,nn),
```

```
        tcl('global einzelschritt ; set einzelschritt 1'),
        tk_loop_while_events.
einzelschritt_aus :-
        unset_global(einzelschritt),
        tcl('global einzelschritt ; set einzelschritt 0'),
        tk_loop_while_events.


/* Bietet dem Benutzer an der augenblicklichen Belegungsstelle die
Auswahl der möglichen Belegungen an.
*/
d_auswahl(_,_,[],_,_) :-
        not current_global(triviale_einzelschritte),
        !,
        fail.

d_auswahl(_,_,[[Muster,Kosten,_]],Muster,Kosten) :-
        not current_global(triviale_einzelschritte),
        !.

d_auswahl(Schwester,Tag,MKM,Muster,Kosten) :-
        tcl_string(MKM,MKMS),
        tcl('d_auswahl ## ## ##',[Schwester,Tag,MKMS],Res),
        append(M,[K],Res),
        (
            Muster=M,
            Kosten=K
        ;
            select(MKM,[M,K,_],MKM_),
            d_auswahl(Schwester,Tag,MKM_,Muster,Kosten)
        ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Debugger
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Plan bearbeiten / Konsistenz pruefen
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

bearbeiten_pruefen(L) :-
        unset_global(konsistenz_pruefen_setzen),
        set_global(konsistenz_pruefen,L).


bearbeiten_setzen(L) :-
        set_global(konsistenz_pruefen_setzen,nn),
        set_global(konsistenz_pruefen,L).


beste_loesung_anzeigen :-
        beste_loesung(_,Plan,_),
        tcl_string(Plan,PlanS),
        tcl('anzeigen ##',[PlanS]).



konsistenz_pruefen(Plan,DS) :-
        beste_loesung(Phase,_,_),
        konsistenz_pruefen(Phase,Plan,DS),
        ( current_global(konsistenz_pruefen_setzen) ->
            unset_global(konsistenz_pruefen_setzen),
            tcl('bearbeiten_gesetzt')
        ; true ),
        tcl(konsistenz_darst),
        !.
konsistenz_pruefen(_,_).


konsistenz_pruefen(0,_Plan,_DS).
konsistenz_pruefen(Phase,Plan,DS) :-
        konsistenz_pruefen(Phase,DS,Plan,Plan_),
        Phase_ is Phase-1,
        konsistenz_pruefen(Phase_,Plan_,DS),
        ( current_global(konsistenz_pruefen_setzen) ->
            retractall(loesung(Phase,_,_)),
            asserta(loesung(Phase,Plan,_))
        ; true ).


konsistenz_pruefen(Phase,DS,Plan,Plan_) :-
        dsSW31(DS,SW31),
        unset_global(konsistent),
```

```
        once(bearbeitung c belegen_plan(Phase,1,SW31,_Kosten,Plan_,Plan)),
        once(constraints_eintragen(Phase,Plan_,DS)),
        set_global(konsistent,Plan_),
        fail.
konsistenz_pruefen(_Phase,_DS,_Plan,Plan_) :-
        current_global(konsistent),
        get_global(konsistent,Plan_),
        unset_global(konsistent).



/* IM PRINZIP:


plan_erzeugen(Plan3) :-
        var(Plan3),

        Plan0 = [.,.,.],

        variablen_erzeugen(Vars1),
        constraints(phase(1),Plan0,Vars1),
        belegen(phase(1),Plan0,Vars1,Plan1), % Plan0 -> Plan1

        variablen_erzeugen(Vars2),
        constraints(phase(2),Plan1,Vars2),
        belegen(phase(2),Plan1,Vars2,Plan2), % Plan1 -> Plan2

        variablen_erzeugen(Vars3),
        constraints(phase(3),Plan2,Vars3),
        belegen(phase(3),Plan2,Vars3,Plan3). % Plan2 -> Plan3



konsistent(Plan3) :-
        not var(Plan3),

        variablen_erzeugen(Vars3),
        belegen(phase(3),Plan2,Vars3,Plan3), % Plan3 -> Plan2
        constraints(phase(3),Plan2,Vars3),

        variablen_erzeugen(Vars2),
        belegen(phase(2),Plan1,Vars2,Plan2), % Plan2 -> Plan1
        constraints(phase(2),Plan1,Vars2),

        variablen_erzeugen(Vars1),
        belegen(phase(1),Plan0,Vars1,Plan1), % Plan1 -> Plan0
        constraints(phase(1),Plan0,Vars1),

        Plan0 = [.,.,.].

*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plan bearbeiten / Konsistenz pruefen
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Inkonsistenzen darstellen
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Anzeige von Inkonsistenzen */

%inkonsistent(Id,C) :-


inkonsistent(bearbeitung,_) :-
        fehler([
            'Fehler:\\n\\n',
            'Der Dienstplan enthält einen unbekannten Eintrag.']).


inkonsistent((belegung,Phase,Tag),(_ ?>= Min)) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
            '\\nINKONSISTENZ:\\n\\n',
            'Am Tag ',Tag,' kann die minimale Belegung (',Min,
            ') nicht eingetragen werden.']).

inkonsistent((belegung,Phase,Tag),(_ ?=< Max)) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
```

```
            '\\nINKONSISTENZ:\\n\\n',
            'Am Tag ',Tag,' kann die maximale Belegung (',Max,
            ') nicht eingetragen werden.']).


inkonsistent((einschraenkung,Name,Phase,Schwester,Tag),_) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
            'Belegungsgrenzen pro Tag eingetragen.','\\n',
            '\\nINKONSISTENZ:\\n\\n',
            'Einschränkung ''',Name,''' kann für Schwester ',Schwester,
            ' am Tag ',Tag,' nicht eingetragen werden.']).


inkonsistent((Schwester,Tag),wunsch(Phase,Wunsch,_,_)) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
            'Belegungsgrenzen pro Tag eingetragen.','\\n',
            'Globale Einschränkungen eingetragen.','\\n',
            '\\nINKONSISTENZ:\\n\\n',
            'Am Tag ',Tag,' kann für die Schwester ',Schwester,
            ' der Wunsch ''',Wunsch,''' nicht eingetragen werden.']).


inkonsistent((bereich,Phase,Schwester),(_ ?>= Min)) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
            'Belegungsgrenzen pro Tag eingetragen.','\\n',
            'Globale Einschränkungen eingetragen.','\\n',
            'Wünsche eingetragen.','\\n',
            '\\nINKONSISTENZ:\\n\\n',
            'Für Schwester ',Schwester,' kann die minimale Belegung (',Min,
            ') nicht eingetragen werden.']).

inkonsistent((bereich,Phase,Schwester),(_ ?=< Max)) :-
        fehler([
            'Phase ',Phase,':\\n\\n',
            'Bisheriger Plan eingetragen.','\\n',
            'Belegungsgrenzen pro Tag eingetragen.','\\n',
            'Globale Einschränkungen eingetragen.','\\n',
            'Wünsche eingetragen.','\\n',
            '\\nINKONSISTENZ:\\n\\n',
            'Für Schwester ',Schwester,' kann die maximale Belegung (',Max,
            ') nicht eingetragen werden.']).


inkonsistent(Id,C) :- fehler([Id,C]).
inkonsistent(C) :- fehler([C]).



% Fehler darstellen:

fehler(L) :-
        tcl_string(L,Ls),
        tcl('fehler ##',[Ls]).




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Inkonsistenzen darstellen
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.9 std.pro

Die folgende Datei definiert Prädikate, die von IF/Prolog nicht bereitgestellt werden, aber auch nicht spezifisch für INTERDIP sind.

```
/* std.pro

   Nochn paa Standads

   Hans Schlenker (HS), 28.8.96
*/



get_user_parameter(Name,Param) :-
        user_parameters(List),
        get_user_parameter(Name,List,Param).
```

```
get_user_parameter(Name,[Name,Param|_],Param) :- !.
get_user_parameter(Name,[_|R],Param) :- get_user_parameter(Name,R,Param).



% create_apply(Goal,VarL,Apply) :- "Apply = Goal(VarL)".

create_apply(ylppa(Goal,VarL),VarL_,Goal_) :-
        !,
        copy_term((VarL,Goal),(VarL_,Goal_)).
create_apply(Goal,VarL,Apply) :-
        Goal =.. GD,
        append(GD,VarL,AD),
        Apply =.. AD.



% apply(Goal,VarL) :- "Goal(VarL)".

apply(G,V) :-
        create_apply(G,V,A),
        A.



% map(L1,Goal,L2) :- fuer entsprechende Paare (E1,E2), E1 aus L1, E2 aus L2,
%                    gilt Goal(E1,E2) (s.a.o. apply/2).

map([],_,[]).
map([E1|R1],G,[E2|R2]) :-
        apply(G,[E1,E2]),
        map(R1,G,R2).



%mapper(ListList,VarList,Goal) :-
mapper(LL,VL,G) :-
        transpose(LL,LLt),
        rec_doall(LLt,VL,G).

%mapperi(ListList,VarList,Index=Start,Goal) :-
mapperi(LL,VL,I=S,G) :-
        transpose(LL,LLt),
        rec_doall_i(LLt,VL,I=S,G).

mapper0(LL,VL,I,G) :- mapperi(LL,VL,I=0,G).
mapper1(LL,VL,I,G) :- mapperi(LL,VL,I=1,G).



doall(Goal) :- Goal,fail ; true.



retractall(G) :- doall(retract(G)).



% count(G,N) :- N ist die Anzahl der Erfuellungen von G.
count(G,N) :-
        findall(c,G,L),
        list_length(L,N).



append([],[]).
append([A|As],B) :-
        append(As,B_),
        append(A,B_,B).



% rec_doall(List,Element,Goal) :- rekursives doall
rec_doall([],_,_).
rec_doall([L|Ls],E,Goal) :-
        copy_term((E,Goal),(L,Goal_)),
        Goal_,
        rec_doall(Ls,E,Goal).

% rec_doall_i(List,Element,Index = Start,Goal) :- rekursives doall mit Index
rec_doall_i([],_,_,_).
rec_doall_i([L|Ls],E,I=S,Goal) :-
        copy_term((E,I,Goal),(L,S,Goal_)),
        Goal_,
        S_ is S+1,
        rec_doall_i(Ls,E,I=S_,Goal).

rec_doall_0(List,Elem,Index,Goal) :- rec_doall_i(List,Elem,Index = 0,Goal).
```

```
rec_doall_1(List,Elem,Index,Goal) :- rec_doall_i(List,Elem,Index = 1,Goal).



% collect(List,El,Term,TermL) :- ein Term-Uebersetzer für Listen

collect([],_,_,[]).
collect([L|Ls],E,Term,[Term_|Ts]) :-
        copy_term((E,Term),(E_,Term_)),
        L = E_,
        collect(Ls,E,Term,Ts).



/* transpose(M,Mt) :- M ist eine Liste von Listen und Mt die transponierte
        dazu, z.B.
transpose([[1,2,3],[4,5,6]],[[1,4],[2,5],[3,6]]).
*/

transpose([[]|_],[]) :- !.
transpose(M,Mt) :-
        not var(M),
        transpose(M,MCol,M_),
        transpose(M_,Mt_),
        Mt=[MCol|Mt_].

transpose([],[],[]) :- !.
transpose([[A|As]|R],[A|Bs],[As|R_]) :-
        transpose(R,Bs,R_).



/* timebounded(Goal,Time) :- Goal wird max. Time sec. ausgefuehrt
*/
timebounded(Goal,Time) :-
        signal_control(alarm,on),
        system:alarm(Time),
        context(Goal,[
            signal(alarm) => throw(timeout),
            catch(timeout) => timebounded_timeout]),
        !,
        system:alarm(0),
        signal_control(alarm,off).
timebounded(_,_) :- timebounded_timeout.
```

```
timebounded_timeout :-
        system:alarm(0),
        signal_control(alarm,off),
        !,
        fail.



/* call_determ(Goal,Success) :-
        Goal wird genau einmal ausgefuehrt und
                Success == true, falls erfolgreich,
                Success == fail, falls nicht.
*/

call_determ(Goal,true) :- Goal, !.
call_determ(_,fail).



/* tcl_to_list
   nur fuer die oberste Ebene!
*/
tcl_to_list([],[]) :- !.
tcl_to_list([A|R],[A|R]) :- !.
tcl_to_list(A,[A]).



/* select(+List,?Elem,?List_) :- List_ ist List ohne Elem
*/

% fuers (vorlaeufige) IFProlog 5.1
:- private select/3.

select([Elem|RL],Elem,RL).

select([E|RL],Elem,[E|RL_]) :-
        select(RL,Elem,RL_).


%select(List,From) :-
select([],_).
select([E|Es],From) :-
```

```
        select(From,E,From_),
        select(Es,From_).



/* nth(First,+N,+List,?Elem,?List_) :-
        Elem ist das Nte Element von List und List_ ist List
        ohne Elem. Die Zaehlung beginnt bei First.
*/

nth(First,First,[Elem|R],Elem,R) :- !.
nth(First,N,[E|R],Elem,[E|R_]) :-
        N > First,
        N1 is N-1,
        nth(First,N1,R,Elem,R_).

nth0(A,B,C,D) :- nth(0,A,B,C,D).
nth1(A,B,C,D) :- nth(1,A,B,C,D).



/* nth(First,?N,+List,+Elem) :-
        Elem ist das Nte Element der Liste List. Zaehlung beginnt bei First.
*/

nth(First,First,[Elem|_],Elem).
nth(First,N,[_|R],Elem) :-
        nth(First,N1,R,Elem),
        N is N1+1.

nth0(A,B,C) :- nth(0,A,B,C).
nth1(A,B,C) :- nth(1,A,B,C).



/* insert1(+N,+List,+Elem,?List_) :- List_ ist List mit Elem an der Nten
        Stelle eingefuegt. Zaehlung beginnt bei 1.
*/

insert1(1,List,Elem,[Elem|List]) :- !.
insert1(N,[E|Es],Elem,[E|Es_]) :-
        N > 1,
        N1 is N-1,
        insert1(N1,Es,Elem,Es_).



/* list_gen(List,Len) :- List besteht aus Len Variablen.
*/

list_gen([],0) :- !.
list_gen([_|Rest],Len) :-
        Len > 0,
        Len_ is Len - 1,
        list_gen(Rest,Len_).



/* flatten(List,List_) :- List_ ist die Liste, die alle Elemente aller
        Elemente aller Elemente aller ... von Liste enthaelt. Z.B.
flatten([[a,b],c,[d,[e,f]]],L).
*/

flatten(A,[A]) :- var(A),!.
flatten([],[]) :- !.
flatten([A|R],L) :-
        !,
        flatten(A,A_),
        flatten(R,R_),
        append(A_,R_,L).
flatten(A,[A]).
```

A.10 constr.pro

`constr.pro` definiert Prädikate zur Bearbeitung und Erzeugung von Constraints. Diese sind allgemein verwendbar.

```
/* constr.pro

   Einige Standards fuer die Constraint-Behandlung.
*/


% Laden der FD-Contraints-Bibliothek
:- import(const_domain).
```

```
:- private copy_term/2.
copy_term(A,B) :- c_copy_term(A,B).


/* c_copy_term/2

   Wie copy_term/2, nur dass mit Constraints versehene Variable wie
   instantiierte Var. behandelt werden, also nicht erneuert werden!!!
*/
c_copy_term(T,T_) :-
        c_copy_term_get_const_vars(T,CVs),
        system:copy_term((T,CVs),(T_,CVs)).

c_copy_term_get_const_vars(V,[V]) :-
        var(V),
        is_domain(V),
        !.
c_copy_term_get_const_vars(V,[]) :-
        var(V),
        !.
c_copy_term_get_const_vars([],[]) :- !.
c_copy_term_get_const_vars([A|As],CVs) :-
        !,
        c_copy_term_get_const_vars(A,CVs1),
        c_copy_term_get_const_vars(As,CVs2),
        append(CVs1,CVs2,CVs).
c_copy_term_get_const_vars(Term,CVs) :-
        Term =.. [_F|Ps],
        !,
        c_copy_term_get_const_vars(Ps,CVs).
c_copy_term_get_const_vars(_,[]).




/* const_sum(Liste,Summe) :- Die Elemente (Domain-Vars bzw. Integer) der
        Liste summieren sich zu Summe (Domain-Var oder Integer).
*/

const_sum(List,Sum) :- list_sum(List,SumL), Sum ?= SumL.
```

```
/* list_sum(List,Sum) :-
*/

list_sum([S],S) :- !.
list_sum([],0) :- !.
list_sum([S|Ss],(S + Sum)) :-
        list_sum(Ss,Sum).



/* collect_vars(Liste,Muster,Bedingung,V,Vs) :-
        Vs ist die Liste der V, zu denen das Muster (V, Muster und Bedingung
        haben idR. gemeinsame Variablen) Element der Liste ist
        und Bedingung erfuellt ist.
*/

collect_vars([],_,_,_,[]) :- !.
collect_vars(Liste,Muster,Bedingung,V,Vs) :-
        copy_term((Muster,Bedingung,V),(M_,B_,V_)),
        Liste = [H|T],
        call_determ((M_ = H, B_),S),
        ( S == true ->
                Vs = [V_|Vs_]
        ;
                Vs = Vs_ ),
        collect_vars(T,Muster,Bedingung,V,Vs_).

collect_vars(L,M,V,Vs) :- collect_vars(L,M,true,V,Vs).



/* collect_var_lists(ForAll,Vars,Pattern,Var,VLs) :-
        fuer jede Erfuellung von ForAll werden die Var, zu denen Pattern auf
        Elemente von Vars passt (ForAll, Pattern und Var haben idR gemeinsame
        Variablen) in VLs gesammelt.
*/

collect_var_lists(ForAll,Vars,Pattern,Var,VLs) :-
        findall((Pattern,Var,_),ForAll,L),
        collect_var_lists1(L,Vars,VLs).

collect_var_lists1([],_,[]).
```

```
collect_var_lists1([(Pattern,Var,VL)|Rest],Vars,[VL|VLs]) :-
        collect_vars(Vars,Pattern,Var,VL),
        collect_var_lists1(Rest,Vars,VLs).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Hilfspraedikate fuer die Erzeugung von Variablenlisten
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


erz_vars_dom(PL,Tupel,Dom) :-
        erz_tupel_vars(PL,Tupel,Vars),
        Vars in Dom.

erz_tupel_vars(PredList,Tupel,Vars) :-
        list_length(PredList,PL),
        list_gen(M,PL),
        append(M,[_],M1),
        erz_vars_krit(PredList,M,Krit),
        findall(M1,Krit,Tupel),
        append(M,[A],M2),
        collect_vars(Tupel,M2,A,Vars).


/*erz_vars_krit(PredList,VarList,Krit) :- Krit ist die Konjunktion von
    P(V) fuer entsprechende P aus PredList und V aus VarList
*/

erz_vars_krit([P],[V],K) :-
        K =.. [P,V].
erz_vars_krit([P|Ps],[V|Vs],(K,Ks)) :-
        K =.. [P,V],
        erz_vars_krit(Ps,Vs,Ks).



/* var_tupel_list(V,V_) :- die Darstellung der Variablen ist von einer
        Tupel-Liste in eine Liste von Listen umgewandelt, z.B.:
var_tupel_list([[1,1,A],[1,2,B],[2,1,C]],[1-[1-[A],2-[B]],2-[1-[C]]]).
*/
```

```
var_tupel_list(TL,L) :-
        var_tupel_list([],TL,L).

var_tupel_list(L,[],L).
var_tupel_list(L,[T|Ts],L_) :-
        var_tupel_list_eintr(L,T,L1),
        var_tupel_list(L1,Ts,L_).



/* var_tupel_list_eintr(List,Tupel,List_) :-
*/

var_tupel_list_eintr(L,[Var],[Var|L]) :- !.
var_tupel_list_eintr(List,[I|Rest],List_) :-
        var_tupel_list_neu(I,List,Neu,Next,List_),
        var_tupel_list_eintr(Next,Rest,Neu).

var_tupel_list_neu(I,[],Neu,[],[I-Neu]) :- !.
var_tupel_list_neu(I,[I1-L|R],Neu,Next,[I1-L|R_]) :-
        I1 < I,
        !,
        var_tupel_list_neu(I,R,Neu,Next,R_).
var_tupel_list_neu(I,[I-Next|R],Neu,Next,[I-Neu|R]) :- !.
var_tupel_list_neu(I,[I1-L|R],Neu,I-Neu,[I-Neu,I1-L|R]).



/* var_list_einfach(L,L_) :- L_ ist die Vereinfachung von L (nur noch
        Variablen in Listen)
*/

var_list_einfach(Var,Var) :- var(Var), !.
var_list_einfach([],[]) :- !.
var_list_einfach([H|T],[H_|T_]) :-
        !,
        var_list_einfach(H,H_),
        var_list_einfach(T,T_).
var_list_einfach(_-L,L_) :- var_list_einfach(L,L_).



/* var_list_member(Element,List) :- Element ist Element von List (enstpr.
        dem Standard-Member, nur eben auf Varlisten).
```

```
*/

var_list_member([I],VL) :-
        member(I,VL).
var_list_member([I|Is],VL) :-
        member((I-VLs),VL),
        var_list_member(Is,VLs).




/* ACHTUNG: folgendes include muss hier am Ende stehen, damit die Redefinition
   von z.B. copy_term sich auch da auswirkt!
*/
:- include('std/std.pro').
```

A.11 interdip.tcl

Folgende Tcl/Tk-Datei definiert den Hauptteil der Benutzerschnittstelle.

```
#!/usr/local/bin/wish -f

option readfile interdip-defaults

# Mit dem Script std/neugen generiere ich neue Versionen. Dieses Script
# ersetzt in allen Dateien "s40" durch die neue Nummer, also z.B.
# s99. Folgende Zeile verwendet dies.
set version "[string range {s40} 1 1].[string range {s40} 2 2]"


source std/place-window.tcl
source std/tk.tcl
source debugger.tcl

# farbige Darst. der einzelnen Schichten im Plan
set farben(0) grey
set farben(1) DarkGreen
set farben(2) red
set farben(3) blue
set farben(.) black
set farben() black



# titel setzt den Titel des Hauptfensters .

proc titel {{zusatz {}}} {
    global version

    wm title . "InterDip Version $version$zusatz"
    wm iconname . "InterDip Version $version$zusatz"
}




# hallo bringt den ersten Dialog auf den Bildschirm, von dem aus der
# Benutzer entweder eine Datei laden kann oder interdip wieder verlassen
# kann.

proc hallo {} {
    global tage schwestern
    global version
    global var

    toplevel .hallo
    label .hallo.h -text InterDip -width 25
    label .hallo.v -text "Version $version" -width 25

    button .hallo.n -text Laden -command {
        if {[einstellungen laden .s]} {
            destroy .hallo
        }
    }
#     button .hallo.d -text Standards -command {
#       einstellungen laden "init.s"
#       destroy .hallo
#     }
    button .hallo.e -text Ende -command exit_prolog
    pack .hallo.h .hallo.v .hallo.n .hallo.e\
            -side top -expand true -fill x

    raise .hallo .
    grab .hallo
    tkwait window .hallo
```

```
    prolog "schwestern S"
    set schwestern $var(S)
    prolog "tage T"
    set tage $var(T)
}



# start erzeugt das Menue bzw die Menues und den Dienstplan (via table).

proc start {} {
    global schwestern tage
    global schwesternL tageL
    global .t
    global status
    global kosten_anzeigen
    global zeitschranke erklaerungen
    global md mw mb mdeb mdebl mp
    global var
    global farben
    global testversion

    prolog "zeitschranke ZS"
    set zeitschranke $var(ZS)

    prolog "erklaerungen ES"
    set erklaerungen $var(ES)

    # folgende Einstellung ist intuitiv klarer (in der Bedienung)
    set triviale_einzelschritte 1
    call_prolog "set_global(triviale_einzelschritte,nn)"

    set tageL {}
    for {set i 1} {$i <= $tage} {incr i} {
        lappend tageL $i
    }

    set schwesternL {KarinG Hilde Gerda Gerd Hubert Anna Norbert\
            EddaA EddaB Ina Franz Rex Chris Christa Emil Peter\
            Berta Carla Klaus Herbert}
    if {$schwestern > 20} {
        for {set i 21} {$i <= $schwestern} {incr i} {
            lappend schwesternL schwester$i
        }
    }
    if {$schwestern < 20} {
        set schwesternL [lrange $schwesternL 0 [expr $schwestern-1]]
    }

    # Menueleiste
    frame .m

    menubutton .m.datei -text "Datei" -menu .m.datei.m
    set md [menu .m.datei.m -tearoff 1]

    menubutton .m.wuensche -text "Wünsche" -menu .m.wuensche.m
    set mw [menu .m.wuensche.m -tearoff 1]

    menubutton .m.bereiche -text "Bereiche" -menu .m.bereiche.m
    set mb [menu .m.bereiche.m -tearoff 1]

    menubutton .m.muster -text "Muster" -menu .m.muster.m
    set mm [menu .m.muster.m -tearoff 1]

    menubutton .m.optionen -text "Optionen" -menu .m.optionen.m
    set mo [menu .m.optionen.m -tearoff 1]

    menubutton .m.debugger -text "Debugger" -menu .m.debugger.m
    set mdeb [menu .m.debugger.m -tearoff 1]

    menubutton .m.phasen -text "Phasen" -menu .m.phasen.m
    set mp [menu .m.phasen.m -tearoff 1]

    pack .m.datei .m.wuensche .m.bereiche .m.muster .m.optionen \
            .m.debugger .m.phasen -side left

    $md add command -label "Einstellungen laden"\
            -command "einstellungen laden"
    $md add command -label "Einstellungen speichern"\
            -command "einstellungen speichern"
    $md add separator
    $md add command -label "Dienstplan ausgeben" \
            -command "call_prolog dr_plan"
    $md add command -label "Dienstplan als LaTeX-Tabelle ausgeben" \
            -command "call_prolog dr_plan_l"
    $md add separator
    $md add command -label "Ende" -command exit_prolog
```

```
$mw add command -label "rot" -command "wuensche_erfassen rot"
$mw add command -label "schwarz" -command "wuensche_erfassen schwarz"
$mw add command -label "weiß" -command "wuensche_erfassen weiss"

$mb add command -label "Schwestern: Phase 1 (Frei/Dienst)" \
        -command "bereich 1"
$mb add command -label "Schwestern: Phase 2 (Nachtschichten)" \
        -command "bereich 2"
$mb add command -label "Schwestern: Phase 3 (Frühschichten)" \
        -command "bereich 3"
$mb add separator
$mb add command -label "Tage: Phase 1 (Frei/Dienst)" \
        -command "belegung 1"
$mb add command -label "Tage: Phase 2 (Nachtschichten)" \
        -command "belegung 2"
$mb add command -label "Tage: Phase 3 (Frühschichten)" \
        -command "belegung 3"
$mb add separator
$mb add command -label "Globale Einschränkungen" \
        -command "einschraenkungen"

$mm add command -label "Phase 1" -command "muster 1"
$mm add command -label "Phase 2" -command "muster 2"
$mm add command -label "Phase 3" -command "muster 3"

$mo add check -label "Kosten anzeigen" -command "kosten_fenster"\
        -variable kosten_anzeigen
$mo add check -label "Berechnung verfolgen" -command berechnung_verfolgen\
        -variable berechnung_verfolgen
$mo add separator
$mo add check -label "Legenden" -variable erklaerungen \
        -command legende

$mdeb add check -label "Debugger" -command debugger\
        -variable debugger
$mdeb add check -label "triviale Einzelschritte" \
        -variable triviale_einzelschritte -command {
    if {$triviale_einzelschritte == 1} {
        call_prolog "set_global(triviale_einzelschritte,nn)"
    } else {
        call_prolog "unset_global(triviale_einzelschritte)"
    }
}
$mdeb add command -label "Breakpoint setzen" -command breakpoint_setzen
```

```
set mdebl [menu $mdeb.breakpoint_loeschen -tearoff 0]
$mdeb add cascade -label "Breakpoint löschen" -menu $mdebl
$mdeb add separator
$mdeb add command -label "Plan bearbeiten" -command bearbeiten

$mp add command -label "Zeitschranke: $zeitschranke" -command {
    wert_aendern zeitschranke "Zeitschranke: "
    call_prolog "retract(zeitschranke(_))"
    call_prolog "asserta(zeitschranke($zeitschranke))"
    $mp entryconfigure "Zeitschranke: *" \
            -label "Zeitschranke: $zeitschranke"
}
$mp add separator
$mp add command -label "Alles löschen" -command {
    call_prolog loeschen
    anzeige_loeschen
}
$mp add separator
$mp add command -label "Phase 1 berechnen" -command "phase_berechnen 1"
$mp add command -label "Phase 2 berechnen" -command "phase_berechnen 2"
$mp add command -label "Phase 3 berechnen" -command "phase_berechnen 3"
$mp add separator
$mp add command -label "aktuelle Phase abbrechen" \
        -command berechnung_stoppen

# Tabelle
canvas .t
#    table .t $tageL $schwesternL 2 label platz_gedrueckt
table .t $tageL $schwesternL 2 entry platz_gedrueckt
for {set y 1} {$y <= $schwestern} {incr y} {
    for {set x 1} {$x <= $tage} {incr x} {
        [set .t($x,$y,w)] configure -state disabled
    }
}

sonntage_markieren .t

# Statuszeile
label .s -anchor w -textvariable status
set status {}

pack .m .t .s -side top -anchor w -expand 1 -fill x

legende
```

```
    bind . q {
        exit_prolog
    }

    # falls TTK (prolog time_til_kill(TTK)) größer als 0 ist, ist dies
    # der Aufruf als Testversion und interdip wird nach TTK Sekunden
    # beendet.
    set testversion 0
    prolog "get_global time_til_kill TTK"
    if {$var(TTK) > 0} {
        set testversion 1
        # macht aus Sek. Millisek.
        set TTK [expr $var(TTK)*1000]
        after $TTK {
            tk_messageBox -icon warning -type ok -message \
"In einer Minute wird diese InterDip-Test-Version beendet!"

            after 60000 {
                tk_messageBox -icon error -type ok -message \
"Diese InterDip-Test-Version wird jetzt beendet!"

                exit_prolog
            }
        }
    }
}



# legende erzeugt für alle Fenster die Legenden und löscht sie auch wieder,
# je nachdem.

proc legende {} {
    global erklaerungen farben
    call_prolog "retract(erklaerungen(_))"
    call_prolog "asserta(erklaerungen($erklaerungen))"

    if {$erklaerungen == 0} {
        catch {destroy .e}
        catch {destroy .wuenscherot.e}
        catch {destroy .wuenscheschwarz.e}
        catch {destroy .wuenscheweiss.e}
        catch {destroy .bereich1.e}
        catch {destroy .bereich2.e}
        catch {destroy .bereich3.e}
        catch {destroy .muster1.e}
        catch {destroy .muster2.e}
        catch {destroy .muster3.e}
        catch {destroy .belegung1.e}
        catch {destroy .belegung2.e}
        catch {destroy .belegung3.e}
        catch {destroy .debug.e}
        catch {destroy .bearbeiten.e}
        return
    }
    if {$erklaerungen > 0} {
        catch {
            text .e -height 3 -width 20 -wrap word
            .e tag configure farben0 -foreground $farben(0)
            .e tag configure farben1 -foreground $farben(1)
            .e tag configure farben2 -foreground $farben(2)
            .e tag configure farben3 -foreground $farben(3)
            .e tag configure farben. -foreground $farben(.)
            .e insert end "Legende: '"
            .e insert end 0 farben0
            .e insert end "' Frei(schicht), '"
            .e insert end 1 farben1
            .e insert end "' Frühschicht, '"
            .e insert end 2 farben2
            .e insert end "' Spätschicht, '"
            .e insert end 3 farben3
            .e insert end "' Nachtschicht"
            .e insert end ", '"
            .e insert end . farben.
            .e insert end "' (noch) unbestimmt"

            pack .e -side top -anchor w -expand 1 -fill x
        }

        foreach farbe {rot schwarz weiss} {
            catch {
                text .wuensche$farbe.e -height 3 -width 50 -wrap word
                switch -- $farbe {
                    rot {
                        .wuensche$farbe.e insert end \
"rote Wünsche: unbedingt zu erfüllend (Kosten: 10000)"
                    }
```

```
                    schwarz {
                        .wuensche$farbe.e insert end \
"schwarze Wünsche: wichtige Wünsche
Kosten können geändert werden (ist der Wert grösser oder gleich 10000,\
werden auch diese Wünsche als harte Wünsche behandelt)."
                    }
                    weiss {
                        .wuensche$farbe.e insert end \

"weiße Wünsche: weniger wichtige, 'Standard'-Wünsche
Kosten können geändert werden (ist der Wert grösser oder gleich 10000,\
werden auch diese Wünsche als harte Wünsche behandelt)."
                    }
                }
                pack .wuensche$farbe.e -side top -anchor w -expand 1 -fill x
            }
        }

        foreach nr {1 2 3} {
            catch {
                text .bereich$nr.e -height 4 -width 30 -wrap word
                switch -- $nr {
                    1 {
                        .bereich$nr.e insert end \
"Bereiche pro Schwester in Phase 1:
'Arbeitszeit' (Anzahl aller Schichten)"
                    }
                    2 {
                        .bereich$nr.e insert end \
"Bereiche pro Schwester in Phase 2:
Anzahl der Nachtschichten"
                    }
                    3 {
                        .bereich$nr.e insert end \
"Bereiche pro Schwester in Phase 3:
Anzahl der Frühschichten"
                    }
                }
                pack .bereich$nr.e -side top -anchor w -expand 1 -fill x
            }
        }

        foreach phase {1 2 3} {
            catch {
                text .muster$phase.e -height 3 -width 50 -wrap word
                .muster$phase.e insert end \
"Muster: neuer Eintrag: am Ende anfügen;\
Eintrag löschen: markierte Zeile (die ein markiertes Feld enthält) wird\
gelöscht;\
Pfeil nach oben: Eintrag wird um eine Stelle nach oben geschoben (Muster\
wird dadurch bevorzugt benutzt);\
Pfeil nach unten: Eintrag wird um eine Stelle nach unten verschoben (Muster\
wird dadurch seltener benutzt)"

                pack .muster$phase.e -side top -anchor w -expand 1 -fill x
            }
        }

        foreach nr {1 2 3} {
            catch {
                text .belegung$nr.e -height 4 -width 30 -wrap word
                switch -- $nr {
                    1 {
                        .belegung$nr.e insert end \
"Belegung Phase 1: Anzahl der Schwestern in allen Schichten zusammen\
(wieviele Schwestern an dem Tag insgesamt arbeiten). Eingabe für jeden\
Tag getrennt"
                    }
                    2 {
                        .belegung$nr.e insert end \
"Belegung Phase 2: Anzahl der Schwestern in der Nachtschicht;\
in Abhängigkeit der arbeitenden Schwestern"
                    }
                    3 {
                        .belegung$nr.e insert end \
"Belegung Phase 3: Anzahl der Schwestern in der Frühschicht;\
in Abhängigkeit der Schwestern in Früh- und Spätschicht zusammen"
                    }
                }
                pack .belegung$nr.e -side top -anchor w -expand 1 -fill x
            }
        }

        catch {
            text .debug.e -height 3 -width 30 -wrap word
            .debug.e insert end \
"Tasten: gezielter Rücksprung, einen Schritt rückwärts, Pause (an/aus),\
Weitermachen, Einzelschrittmodus (an/aus)"
```

```
            pack .debug.e -side top -anchor w -expand 1 -fill x
        }

        catch {
            text .bearbeiten.e -height 3 -width 30 -wrap word
            .bearbeiten.e insert end \
"Konsistenz prüfen: prüft den aktuellen Plan auf Korrektheit;\
Plan übernehmen: beendet die Bearbeitung erfolgreich;\
Abbruch: macht die Änderungen rückgängig"
            pack .bearbeiten.e -side top -anchor w -expand 1 -fill x
        }
    }
}



# platz_gedrueckt wird aufgerufen, falls der Benutzer eine Zelle des
# Dienstplans anklickt. Ausserdem wird dann die globale Variable
# platz(gedrueckt) veraendert, worauf eine Prozedur, die das benötigt,
# warten kann (tkwait). In platz(x) und platz(y) stehen dann die
# Koordinaten der "gedrueckten Zelle"

set platz(gedrueckt) 0

proc platz_gedrueckt {x y} {
    global platz
    set platz(x) $x
    set platz(y) $y
    set platz(gedrueckt) [expr 1-$platz(gedrueckt)]
}



# wuensche_erfassen erzeugt für die angegebene Farbe eine
# Eingabetabelle und lädt die Wünsche von Prolog.

proc wuensche_erfassen {farbe} {
    global var
    global tage tageL schwestern schwesternL
    global kosten_dafuer

    set w .wuensche$farbe
    catch "destroy $w"
    toplevel $w
    wm title $w "'${farbe}e' Wünsche"
    frame $w.m
    button $w.m.setzen -text "setzen" -command "wuensche_setzen $farbe"
    button $w.m.schliessen -text "schließen"\
           -command "destroy $w"
    bind $w.m.schliessen <Destroy> "wuensche_setzen $farbe"
    pack $w.m.setzen $w.m.schliessen -side left
    frame $w.t
    pack $w.m $w.t -side top -anchor w -expand true -fill x

    legende

    set fv $w.t
    table $fv $tageL $schwesternL 2 entry
    sonntage_markieren $fv
    global $fv
    prolog "wuensche $farbe KostenDafuer W"
    set kosten_dafuer($farbe) $var(KostenDafuer)
    for {set y 1} {$y <= $schwestern} {incr y} {
        for {set x 1} {$x <= $tage} {incr x} {
            set x0 [expr $x - 1]
            set y0 [expr $y - 1]
            eval set ${fv}($x,$y) \
                    [lindex [lindex $var(W) $y0] $x0]
        }
    }
    button $w.m.k -text "Kosten: $kosten_dafuer($farbe)" \
           -command "
        set wuensche_erfassen_kosten \$kosten_dafuer($farbe)
        wert_aendern wuensche_erfassen_kosten {Kosten dafür: }
        set kosten_dafuer($farbe) \$wuensche_erfassen_kosten
        $w.m.k configure -text \"Kosten: \$kosten_dafuer($farbe)\"
    "
    pack $w.m.k -side right
    if {$farbe == "rot"} {
        $w.m.k configure -state disabled
    }
}


# wuensche_setzen schreibt die angezeigten Wünsche nach Prolog.

proc wuensche_setzen {farbe} {
    global tage schwestern
```

```
    set fv .wuensche$farbe.t
    global $fv
    global kosten_dafuer

    set l {}
    for {set y 1} {$y <= $schwestern} {incr y} {
        set lx {}
        for {set x 1} {$x <= $tage} {incr x} {
            lappend lx "[set ${fv}($x,$y)]"
        }
        lappend l $lx
    }
    prolog "wuensche_setzen $farbe $kosten_dafuer($farbe) {$l}"
}



# sonntage_markieren ändert die Farbe der Spaltenüberschriften einer
# Tabelle, die für einen Samstag, Sonntag usw. stehen, in rot um.

proc sonntage_markieren {tab} {
    global var tage
    prolog "sonntage L"
    if {$var(L) == "{}"} return
    foreach s $var(L) {
        $tab.$s.t configure -foreground red
    }
}



# muster zeigt einen Eingabebildschirm für die Muster der angegebenen
# Phase an und lädt die Muster von Prolog.

proc muster {phase} {
    global var
    global muster

    set w .muster$phase
    catch "destroy $w"
    toplevel $w
    wm title $w "Muster Phase $phase"
    frame $w.m
    button $w.m.setzen -text "setzen" \
        -command "muster_setzen $phase"
    button $w.m.schliessen -text "schließen"\
        -command "destroy $w"
    bind $w.m.schliessen <Destroy> "muster_setzen $phase"
    pack $w.m.setzen $w.m.schliessen -side left
    frame $w.t
    frame $w.b
    button $w.b.neu -text "neuer Eintrag" -command "
        incr muster($phase,LetzteZeile)
        muster_neue_zeile $phase $w.t.\$muster($phase,LetzteZeile)
        pack $w.t.\$muster($phase,LetzteZeile) -side top -anchor w
        lappend muster($phase,Zeilen) $w.t.\$muster($phase,LetzteZeile)
    "
    button $w.b.loeschen -text "Eintrag löschen" -command {
        if {![regexp {\.muster([0-9]+)\.t\.[0-9]+} [focus] zeile phase]} \
                return
        set i [lsearch $muster($phase,Zeilen) $zeile]
        set muster($phase,Zeilen) [lreplace $muster($phase,Zeilen) $i $i]
        destroy $zeile
    }
    pack $w.b.neu $w.b.loeschen -side left
    pack $w.m $w.t $w.b -side top -anchor w

    prolog "muster_holen $phase M"

    # Die Var. eines frueheren Muster-setzen-Fensters entfernen
    set vs [array names muster "$phase,*"]
    foreach v $vs {
        unset muster($v)
    }
    set muster($phase,Zeilen) {}
    set muster($phase,LetzteZeile) 0

    foreach m $var(M) {
        set zeile $w.t.[incr muster($phase,LetzteZeile)]
        set muster($phase,$zeile,Kosten) [lindex $m 1]
        set i 0
        foreach b [lindex $m 0] {
            set muster($phase,$zeile,$i) $b
            incr i
        }
        muster_neue_zeile $phase $zeile
        lappend muster($phase,Zeilen) $zeile
    }
```

```
    eval pack $muster($phase,Zeilen) -side top -anchor w

    legende
}


# muster_neue_zeile fügt am Ende der Musterliste eine neue Zeile an.

proc muster_neue_zeile {phase zeile} {
    global muster

    frame $zeile
    button $zeile.up -image up -command "
        set i \[lsearch \$muster($phase,Zeilen) $zeile\]
        if {\$i > 0} {
            pack forget  \[lindex \$muster($phase,Zeilen) \$i\]
            pack \[lindex \$muster($phase,Zeilen) \$i\] \
                    -before \[lindex \$muster($phase,Zeilen) \[expr \$i-1\]\]
            set muster($phase,Zeilen) \[linsert \[lreplace \
                    \$muster($phase,Zeilen) \$i \$i\] \[expr \$i-1\] $zeile\]
        }
    "
    button $zeile.down -image down -command "
        set i \[lsearch \$muster($phase,Zeilen) $zeile\]
        if {\$i < \[expr \[llength \$muster($phase,Zeilen)\]-1\]} {
            pack forget  \[lindex \$muster($phase,Zeilen) \$i\]
            pack \[lindex \$muster($phase,Zeilen) \$i\] \
                    -after \[lindex \$muster($phase,Zeilen) \[expr \$i+1\]\]
            set muster($phase,Zeilen) \[linsert \[lreplace \
                    \$muster($phase,Zeilen) \$i \$i\] \[expr \$i+1\] $zeile\]
        }
    "
    entry $zeile.kosten -width 4 -textvariable muster($phase,$zeile,Kosten)
    frame $zeile.f -width 8
    pack $zeile.up $zeile.down $zeile.kosten $zeile.f -side left
    for {set i 0} {$i < 31} {incr i} {
        entry $zeile.$i -width 2 -textvariable muster($phase,$zeile,$i)
        pack $zeile.$i -side left
    }
}


# muster_setzen schreibt die angezeigten Muster in der Reihenfolge
# nach Prolog.

proc muster_setzen {phase} {
    global muster

    set l {}
    foreach zeile $muster($phase,Zeilen) {
        set lx {}
        set letzte_belegung -1
        for {set i 0} {$i < 31} {incr i} {
            set x $muster($phase,$zeile,$i)
            if {$x == {}} {
                lappend lx "_"
            } else {
                set letzte_belegung $i
                lappend lx $x
            }
        }
        set lx [lrange $lx 0 $letzte_belegung]
        if {$lx != {}} {
            lappend lx $muster($phase,$zeile,Kosten)
            lappend l $lx
        }
    }
    prolog "muster_setzen $phase {$l}"
}



# phase_berechnen stösst in Prolog die Berechnung einer Phase an
# (siehe program/0).

proc phase_berechnen {phase} {
    if {[string match "success" [call_prolog "current_global(phase)"]]} \
            return
    call_prolog "set_global(phase,$phase)"
}


# berechnung_stoppen bricht die Berechnung der aktuellen Phase ab.

proc berechnung_stoppen {} {
    call_prolog "process_events_stop"
}
```

```
# anzeigen zeigt einen Dienstplan an (mit den entsprechenden Farben
# aus farben().

proc anzeigen {liste} {
    global schwestern tage
    global .t
    global farben

    for {set y 1} {$y <= $schwestern} {incr y} {
        for {set x 1} {$x <= $tage} {incr x} {
            set x0 [expr $x - 1]
            set y0 [expr $y - 1]
            set w [lindex [lindex $liste $y0] $x0]
            set .t($x,$y) $w
            [set .t($x,$y,w)] configure -foreground $farben($w) -bg #d9d9d9
        }
    }
}


# anzeige_loeschen macht alle Zellen des Dienstplans leer.

proc anzeige_loeschen {} {
    global schwestern tage
    global .t
    for {set y 1} {$y <= $schwestern} {incr y} {
        for {set x 1} {$x <= $tage} {incr x} {
            set .t($x,$y) {}
            [set .t($x,$y,w)] configure -bg #d9d9d9
        }
    }
}



# statuszeile zegt den angegebenen text zusätzlich in der Statuszeile
# an.

proc statuszeile {text} {
    global status
    append status "$text "
}
```

```
# statuszeile_ruecksetzen verkürzt den Inhalt der aktuellen
# Statuszeile um das letzte Wort.

proc statuszeile_ruecksetzen {} {
# $status um das letzte Wort verkuerzen
    global status
    regexp {^(([^ ]+ )*)[^ ]+ $} $status _ status
}



# kosten_fenster erzeugt das fenster, in dem die Kosten angezeigt
# werden.

proc kosten_fenster {} {
    global kosten_anzeigen
    global schwestern .kosten
    if {$kosten_anzeigen == 1} {
        catch {destroy .kosten}
        toplevel .kosten
        for {set i 1} {$i <= $schwestern} {incr i} {
            label .kosten.$i -bd 4 -textvariable .kosten($i)
            pack .kosten.$i -side top -anchor e -expand 1 -fill x
        }
    } else {
        catch {destroy .kosten}
    }
}


# kosten_anzeigen zeigt die Kosten im Kostenfenster an.

proc kosten_anzeigen {kosten} {
    global .kosten kosten_anzeigen
    if {$kosten_anzeigen == 1} {
        set i 1
        foreach k $kosten {
            set .kosten($i) $k
            incr i
        }
    }
}
```

```
# bereich zeigt ein Fenster zur Eingabe der Belegungsgrenzen pro
# Schwester (Prolog-Fakten bereich/4) an und holt diese von Prolog.

proc bereich {phase} {
    global schwestern schwesternL var

    catch "destroy .bereich$phase"
    toplevel .bereich$phase
    wm title .bereich$phase "Bereiche Phase $phase"
    frame .bereich$phase.b
    button .bereich$phase.b.setzen -text setzen\
            -command "bereich_setzen $phase"
    button .bereich$phase.b.schliessen -text "schließen"\
            -command "destroy .bereich$phase"
    bind .bereich$phase.b.schliessen <Destroy> "bereich_setzen $phase"
    pack .bereich$phase.b.setzen .bereich$phase.b.schliessen -side left
    frame .bereich$phase.t
    prolog "bereich_holen $phase L"
    for {set i 1} {$i <= $schwestern} {incr i} {
        set i0 [expr $i-1]
        frame .bereich$phase.t.$i
        label .bereich$phase.t.$i.l -anchor w -width 10\
                -text [lindex $schwesternL $i0]
        set min [lindex [lindex $var(L) $i0] 0]
        set max [lindex [lindex $var(L) $i0] 1]
        scale_minmax .bereich$phase.t.$i.s hor 31 15 20 .bereich$phase.t.$i\
                "{0 $min 31} {0 $max 31}"
        global .bereich$phase.t.$i
        pack .bereich$phase.t.$i.l .bereich$phase.t.$i.s -side left
        pack .bereich$phase.t.$i -side top -anchor w
    }
    pack .bereich$phase.b .bereich$phase.t -side top -anchor w

    legende
}


# bereich_setzen schreibt die Belegungsgrenzen pro Schwester
# (bereich/4) nach Prolog.

proc bereich_setzen {phase} {
    global schwestern var
    set l {}
    for {set i 1} {$i <= $schwestern} {incr i} {
        global .bereich$phase.t.$i
        lappend l "[set .bereich$phase.t.${i}(min)]\
                [set .bereich$phase.t.${i}(max)]"
    }
    prolog "bereich_setzen $phase {$l}"
}



# einschraenkungen zeigt ein Fenster zum Ein- und Ausschalten aller
# definierten Einschränkungen an und holt die Schalter von Prolog.

proc einschraenkungen {} {
    global einschraenkungen var

    catch "destroy .einschraenkungen"
    toplevel .einschraenkungen
    wm title .einschraenkungen "Globale Einschränkungen"
    frame .einschraenkungen.b
    button .einschraenkungen.b.setzen -text setzen \
            -command "einschraenkungen_setzen"
    button .einschraenkungen.b.schliessen -text "schließen"\
            -command "destroy .einschraenkungen"
    bind .einschraenkungen.b.schliessen <Destroy> "einschraenkungen_setzen"
    pack .einschraenkungen.b.setzen .einschraenkungen.b.schliessen -side left
    frame .einschraenkungen.t
    prolog "einschraenkungen_holen L"
    set i 1
    foreach l $var(L) {
        set einschraenkungen(anzahl) $i
        frame .einschraenkungen.t.$i
        set einschraenkungen($i) [lindex $l 0]
        checkbutton .einschraenkungen.t.$i.c -offvalue aus -onvalue ein \
                -variable einschraenkungen($i)
        label .einschraenkungen.t.$i.l -width 50 -text [lindex $l 1] \
                -anchor w
        label .einschraenkungen.t.$i.p -text "[lindex $l 2]. Phase"
        pack .einschraenkungen.t.$i.c .einschraenkungen.t.$i.l \
                .einschraenkungen.t.$i.p -side left
        pack .einschraenkungen.t.$i -side top -expand 1 -fill x
        incr i
```

```
    }
    pack .einschraenkungen.b .einschraenkungen.t -side top -anchor w
}


# einschraenkungen_setzen schreibt die Schalter der Einschränkungen
# nach Prolog.

proc einschraenkungen_setzen {} {
    global einschraenkungen var
    set l {}
    for {set i 1} {$i <= $einschraenkungen(anzahl)} {incr i} {
        lappend l $einschraenkungen($i)
    }
    prolog "einschraenkungen_setzen {$l}"
}



# belegung zeigt ein Fenster zur Eingabe der Belegungsgrenzen pro
# Tag (Prolog-Fakten belegung/4) an und holt diese von Prolog.

proc belegung {phase} {
    global schwestern var
    global .belegung$phase

    catch "destroy .belegung$phase"
    toplevel .belegung$phase
    wm title .belegung$phase "Belegung Phase $phase"

    frame .belegung$phase.b
    button .belegung$phase.b.setzen -text setzen\
            -command "belegung_setzen $phase"
    button .belegung$phase.b.schliessen -text "schließen"\
            -command "destroy .belegung$phase"
    bind .belegung$phase.b.schliessen <Destroy> "belegung_setzen $phase"
    pack .belegung$phase.b.setzen .belegung$phase.b.schliessen -side left
    frame .belegung$phase.t
    prolog "belegung_holen $phase L"

    if {$phase == 1} {
        set dir vert
    } else {
        set dir hor
    }
    set .belegung$phase {}
    foreach b $var(L) {
        set i [lindex $b 0]
        lappend .belegung$phase $i
        frame .belegung$phase.t.$i
        label .belegung$phase.t.$i.l -width 2 -text $i
        scale_minmax .belegung$phase.t.$i.s $dir $schwestern 15 20 \
                .belegung$phase.t.$i \
                "{0 [lindex $b 1] $schwestern} {0 [lindex $b 2] $schwestern}"
        global .belegung$phase.t.$i
        if {$dir == "hor"} {
            pack .belegung$phase.t.$i.l .belegung$phase.t.$i.s -side left
            pack .belegung$phase.t.$i -side top -anchor w
        } else {
            pack .belegung$phase.t.$i.l .belegung$phase.t.$i.s -side bottom
            pack .belegung$phase.t.$i -side left -anchor s
        }
    }
    pack .belegung$phase.b .belegung$phase.t -side top -anchor w

    legende
}


# belegung_setzen schreibt die Belegungsgrenzen pro Tag
# (belegung/4) nach Prolog.

proc belegung_setzen {phase} {
    global schwestern var
    global .belegung$phase

    set l {}
    foreach i [set .belegung$phase] {
        global .belegung$phase.t.$i
        lappend l "$i [set .belegung$phase.t.${i}(min)]\
                [set .belegung$phase.t.${i}(max)]"
    }
    prolog "belegung_setzen $phase {$l}"
}



# Folgendes ist eine Spezialität für eine Maschine, auf der nur die
```

```
# Tk-Version 4.1 installiert war. Da sind tk_getOpenFile und
# tk_getSaveFile nicht vorhanden, weshalb diese hier trivial definiert
# werden.

if {[exec hostname] == "pgts0019"} {
# auf der procyon ist nur die Tk-Version 4.1 installiert...
    proc tk_getOpenFile args {return "/home1/users/prolog/schwester/init.s"}
    proc tk_getSaveFile args {return "/home1/users/prolog/schwester/init.s"}
}



# einstellungen lädt oder speichert, je nach modus, eine
# Einstellungsdatei. Das ist eine Datei mit Prolog-fakten, die das
# Dienstplanproblem spezifizieren. Siehe laden/1 und speichern/1.

proc einstellungen {modus {fname {}}} {
    # modus ist entweder "laden" oder "speichern" oder bel. Abkuerzungen

    global schwestern tage
    global testversion

    set ret 0
    switch -glob -- $modus \
            l* { set p_com laden ; set tk_com tk_getOpenFile }\
            s* {
        if $testversion {
            tk_messageBox -icon error -type ok -message \
"In der InterDip-Test-Version können Sie nichts speichern!"

            return {}
        }
        set p_com speichern
        set tk_com tk_getSaveFile
    }

    switch -glob -- $fname \
            "" { set suffix .$schwestern.$tage.s } \
            .* { set suffix $fname ; set fname "" } \
            *  { set suffix "" }

    if {$fname == ""} {
        set fname [$tk_com -defaultextension "$suffix"\
                -filetypes "{{Einstellungen} {$suffix}}"]
    }
    if {$fname != ""} {
        if {![string match "*$suffix" "$fname"]} {
            append fname "$suffix"
        }
        set ret [string match "success" [call_prolog "${p_com}('$fname')"]]
        titel " - $fname"
    }
    return $ret
}



# wert_aendern zeigt ein Dialogfeld, in dem der Wert der angegebenen
# Variable vom Benutzer geändert werden kann.

proc wert_aendern {var {l {}}} {
    global $var
    catch "destroy .wert"
    toplevel .wert
    wm title .wert "Wert ändern"
    if {$l == {}} "set l $var"
    label .wert.l -text $l
    entry .wert.e -textvariable $var
    pack .wert.l .wert.e -side left
    focus .wert.e
    bind .wert.e <Return> "destroy .wert"
    grab .wert
    tkwait window .wert
}



# berechnung_verfolgen schaltet das Verfolgen der Berechnung einer
# Phase in Prolog ein oder aus.

proc berechnung_verfolgen {} {
    global berechnung_verfolgen
    if {$berechnung_verfolgen == 1} {
        call_prolog "retract((verfolgen(_,_) :- !))"
    } else {
        call_prolog "asserta((verfolgen(_,_) :- !))"
    }
}
```

```
# verfolgen zeigt den Fortschritt der Berechnung einer Phase, das
# heisst, wo sich die Berechnung der Phase befindet.

proc verfolgen {schwester tag {farbe #d9d9d9}} {
    global .t
    [set .t($tag,$schwester,w)] configure -bg $farbe
}


# schnappen zeigt nicht nur, wo sich die Berechnung befindet, sondern
# auch mit welchem aktuellen Wert. Das ist langsamer als verfolgen und
# wird deshalb nur mit dem Debugger verwendet.

proc schnappen {schwester tag {inhalt {}}} {
    global .t farben
    set .t($tag,$schwester) $inhalt
    [set .t($tag,$schwester,w)] configure -foreground $farben($inhalt)\
            -bg #d9d9d9
}



# bearbeiten lässt den Benutzer einen erzeugten Dienstplan im
# Hauptfenster bearbeiten und zeigt einen Dialog an, in dem Benutzer
# u.a. Konsistenzprüfung aufrufen kann.

proc bearbeiten {} {
    global schwestern tage .t

    if {[string match "success" [call_prolog "current_global(phase)"]]} return

    set w .bearbeiten
    catch "destroy $w"
    toplevel $w
    wm title $w "Plan bearbeiten"
    button $w.pruefen -text "Konsistenz prüfen" \
            -command "bearbeiten_pl bearbeiten_pruefen"
    button $w.schliessen -text "Plan übernehmen"\
            -command "bearbeiten_pl bearbeiten_setzen"
    button $w.abbruch -text "Abbruch"\
            -command bearbeiten_gesetzt
    bind $w.abbruch <Destroy> {
        for {set y 1} {$y <= $schwestern} {incr y} {
            for {set x 1} {$x <= $tage} {incr x} {
                [set .t($x,$y,w)] configure -state disabled
            }
        }
    }
    pack $w.pruefen $w.schliessen $w.abbruch -side top -expand true -fill x
    legende

    for {set y 1} {$y <= $schwestern} {incr y} {
        for {set x 1} {$x <= $tage} {incr x} {
            [set .t($x,$y,w)] configure -state normal
        }
    }
}


# bearbeiten_gesetzt zeigt den nach Prolog geschriebenenen Dienstplan
# nochmal (mit den korrektan Farben) an.

proc bearbeiten_gesetzt {} {
    call_prolog beste_loesung_anzeigen
    destroy .bearbeiten
}


# bearbeiten_pl prüft den angezeigten Dienstplan auf Konsistenz oder
# schreibt ihn nach Prolog, je nach p_com.

proc bearbeiten_pl {p_com} {
    global schwestern tage .t

    set l {}
    for {set y 1} {$y <= $schwestern} {incr y} {
        set ly {}
        for {set x 1} {$x <= $tage} {incr x} {
            lappend ly [set .t($x,$y)]
        }
        lappend l $ly
    }
    prolog "$p_com {$l}"
}
```

```
# fehler zeigt den (als Liste l) angegebenen Text dem Benutzer in
# einem trivialen Dialogfeld.

proc fehler {l} {
    global schwesternL

    set msg [join $l {}]

    # ein kleiner Hack zur Verbesserung der Darstellung durch Nennung
    # des Namens der betr. Schwester, ohne dass Prolog diese kennen muss
    if {[regexp {Schwester ([0-9]+)} $msg subex nr] == 1} {
        regsub {Schwester [0-9]+} $msg \
                "$subex ([lindex $schwesternL [expr $nr-1]])" msg
    }

    tk_messageBox -title Fehler -icon error -type ok \
            -message $msg
}

proc konsistenz_darst {} {
    tk_messageBox -title konsistent -icon info -type ok \
            -message "Der Plan ist korrekt."
}



# hallo und start sind die einzigen Prozeduern, die zur
# Initialisierung direkt aufgerufen werden. Der ganze Rest wird von
# der Benutzerschnittstelle durch die Menüs aufgerufen.

hallo
start
```

A.12 debugger.tcl

`debugger.tcl` enthält die Prozeduren, die den Debugger implementieren.

```
# debugger.tcl
# INTERDIPs Debugger


# Debugger ein- und ausschslten

proc debugger {} {
    global debugger
    # Falls die Berechnung gerade laeuft, kann der Deb. weder
    # ein- noch ausgeschaltet werden
    if {[string match "success" [call_prolog "current_global(phase)"]] == 1} {
        set debugger [expr 1-$debugger]
        return
    }
    if {$debugger == 1} {
        debug_ein
    } else {
        catch {destroy .debug}
        call_prolog "debug_aus"
    }
}


# Debug-Fenster erzeugen

proc debug_ein {} {
    global pause einzelschritt
    global debugger

    call_prolog "debug_ein"
    catch {destroy .debug}
    toplevel .debug
    wm title .debug Debugger
    wm protocol .debug WM_DELETE_WINDOW {
        set debugger 0
        debugger
    }

    frame .debug.b
    button .debug.b.fb -image fastbackward -command {
        catch {destroy .auswahl}
        if {$pause == 0} {
            call_prolog pause_ein
        }
        zurueckspringen
    }
```

```
    button .debug.b.b -image backward -command {
        catch {destroy .auswahl}
        if {$pause == 0} {
            call_prolog pause_ein
        }
        call_prolog "set_global(zurueck,(10000,10000))"
    }
    checkbutton .debug.b.pause -text Pause \
            -variable pause -relief raised -padx 8 -pady 4 \
            -command {
        if {$pause == 1} {
            call_prolog pause_ein
        } else {
            call_prolog pause_aus
        }
    }
    button .debug.b.ff -image fastforward -command {
        catch {destroy .auswahl}
        if {$einzelschritt == 1} {
            call_prolog einzelschritt_aus
        }
        if {$pause == 1} {
            call_prolog pause_aus
        }
    }
    checkbutton .debug.b.einzelschritt -text Einzelschritt \
            -variable einzelschritt -relief raised -padx 8 -pady 4 \
            -command {
        if {$einzelschritt == 1} {
            call_prolog einzelschritt_ein
        } else {
            call_prolog einzelschritt_aus
        }
    }
    pack .debug.b.fb .debug.b.b .debug.b.pause .debug.b.ff \
            .debug.b.einzelschritt -side left
    pack .debug.b

    legende
}


proc breakpoint_setzen {} {
    global platz
    global .t
    global mdebl tageL schwesternL
    statuszeile BreakpointAnklicken...
    tkwait variable platz(gedrueckt)
    set x $platz(x)
    set y $platz(y)
    if {[string match "fail" [call_prolog "breakpoint($y,$x)"]]} {
        call_prolog "asserta(breakpoint($y,$x))"
        [set .t($x,$y,w)] configure -bg orange
        set l "Schwester [lindex $schwesternL [expr $y-1]],\
                Tag [lindex $tageL [expr $x-1]]"
        #puts $l
        $mdebl add command -label "$l" \
                -command "breakpoint_loeschen $x $y {$l}"
    }
    statuszeile_ruecksetzen
}

proc breakpoint_loeschen {x y l} {
    global mdebl
    global .t
    call_prolog "retract(breakpoint($y,$x))"
    [set .t($x,$y,w)] configure -bg #d9d9d9
    $mdebl delete $l
}


# gezielter Ruecksprung im Dienstplan, mit Rueckgaengigmachen der Belegung

proc zurueckspringen {} {
    global platz
    global .t
    statuszeile RuecksprungpunktAnklicken...
    tkwait variable platz(gedrueckt)
    statuszeile_ruecksetzen
    call_prolog "set_global(zurueck,($platz(y),$platz(x)))"
}


# Auswahl-Fenster des Debuggers

proc d_auswahl {schwester tag l} {
    global d_auswahl
```

```
    global .t
    catch {destroy .auswahl}
    toplevel .auswahl
    wm title .auswahl Auswahl
    set i 0
    [set .t($tag,$schwester,w)] configure -bg orange
    if {$l != {}} {
        set a [lindex $l 0]
        # Standarwert, falls das Fenster geschlossen wird
        set d_auswahl "[lindex $a 0] [lindex $a 1]"
    }
    foreach a $l {
        frame .auswahl.$i
        button .auswahl.$i.b -text [lindex $a 2] \
                -command "
            set d_auswahl {[lindex $a 0] [lindex $a 1]}
            destroy .auswahl
        "
        label .auswahl.$i.k -text [lindex $a 1]
        pack .auswahl.$i.b .auswahl.$i.k -side left
        pack .auswahl.$i -side top -anchor w
        incr i
    }
    button .auswahl.fail -text fail \
            -command "set d_auswahl fail ; destroy .auswahl"
    pack .auswahl.fail -side top -expand true -fill x
    place_window .auswahl
    tkwait window .auswahl
    return "$d_auswahl"
}
```

A.13 tk.tcl

`tk.tcl` enthält vor allem Prozeduren zur Erzeugung einer Tabelle (zum Beispiel für einen Dienstplan), und Eingabe-Reglern.

```
# Standards fuer TclTk

source std/bitmaps.tcl

# table erzeugt eine Tabelle, wie sie z.B. fuer die Anzeige des
# Dienstplans verwendet wird. Dazu wird der Name eines Widgets (w)
# uebergeben, und zwei Listen, die die Beschriftungen in x-Richtung
# und in y-Richtung enthalten. width ist die Breite einer Zelle, el
# entweder label oder entry das dann für die Zelle erzeugt wird und
# bindung der Name einer Prozedur, die an jede Zelle gabunden wird und
# beim Klicken mit der Maus aufgreufen wird, mit der x- und y-
# Koordinate der Zelle als Parameter. Der Inhalt jeder Zelle wird an
# eine globale Variable eines Arrays gebunden, dessen Name der des
# Widgets ist.

proc table {w xl yl {width 5} {el label} {binding {}}} {
    global $w
    frame $w.t
    label $w.t.t -width $width -bd 4
    pack $w.t.t -in $w.t -side top
    for {set y 1} {$y <= [llength $yl]} {incr y} {
        label $w.t.$y -text [lindex $yl [expr $y-1]] -bd 4
        pack $w.t.$y -in $w.t -side top -anchor w
    }
    pack $w.t -in $w -side left
    for {set x 1} {$x <= [llength $xl]} {incr x} {
        frame $w.$x
        label $w.$x.t -width $width -text [lindex $xl [expr $x-1]] -bd 4
        pack $w.$x.t -in $w.$x -side top
        for {set y 1} {$y <= [llength $yl]} {incr y} {
            switch -glob -- $el \
                    e* "entry $w.$x.$y -width $width -justify center\
                        -textvariable ${w}($x,$y)" \
                    l* "label $w.$x.$y -width $width -justify center\
                        -textvariable ${w}($x,$y) -bd 4 -relief sunken"
            set ${w}($x,$y,w) $w.$x.$y
            pack $w.$x.$y -in $w.$x -side top
            if {$binding != {}} {
                bind $w.$x.$y <Button-1> "$binding $x $y"
            }
        }
        pack $w.$x -in $w -side left
    }
}
```

```
# "Scaler"
# Am Ende der Datei befindet sich ein Test-Beispiel.
#
# Ausserdem gibts hier noch scale_minmax: siehe unten.
#
# Hans Schlenker (HS), 8.1.97

# Wenn ich mich richtig erinnere, habe ich beim Enstehen dieser
# Proz. intensiv scaler2.tcl von Brent Welch benutzt, das ein Beispiel
# aus seinem Tcl/Tk-Einfuehrung ist.

# horizontale "Scaler" sind von links nach rechts ausgerichtet,
# vertikale von oben nach unten.


proc scaler {w dir len grid width var doms} {
    global scaler
    global $var

# damit sind Abkuerzungen moeglich: vert..., vert, ver, ve, v,
# hor..., hor, ho, h
    set dir [string index $dir 0]

    set scaler($w,len) $len
    set scaler($w,grid) $grid
    set scaler($w,width) $width
    set domains [llength $doms]
    set m $len
    for {set i 0} {$i < $domains} {incr i} {
        set d [lindex $doms $i]
        if {[llength $d] == 1} {
            set scaler($w,$i,min) 0
            set scaler($w,$i,max) 1000000
            if {$i < $domains-1} {
                set scaler($w,$i,val) $d
            } else {
                set scaler($w,$i,val) $m
            }
        } else {
            set scaler($w,$i,min) [lindex $d 0]
            set scaler($w,$i,max) [lindex $d 2]
            if {$i < $domains-1} {
                set scaler($w,$i,val) [lindex $d 1]
            } else {
                set scaler($w,$i,val) $m
            }
        }
        set m [expr $m-$scaler($w,$i,val)]
        set ${var}($i) $scaler($w,$i,val)
    }
    set scaler($w,domains) $domains
    set scaler($w,var) $var

    if {$dir == "h"} {
        canvas $w -width [expr ($len+2)*$grid] -height $width
        $w create rect [expr $grid/2] 0 [expr ($len+1.5)*$grid] $width \
                -outline DarkGray -fill white
    } else {
        canvas $w -width $width -height [expr ($len+2)*$grid]
        $w create rect 0 [expr $grid/2] $width [expr ($len+1.5)*$grid] \
                -outline DarkGray -fill white
    }
    set scalers [expr $domains-1]
    set m 0
    for {set i 0} {$i < $scalers} {incr i} {
        incr m $scaler($w,$i,val)
        set scaler($w,$i,pos) $m
        if {$dir == "h"} {
            $w create rect [expr ($m+0.5)*$grid] 0 \
                    [expr ($m+1.5)*$grid] $width \
                    -fill DarkGray -outline black -tag "scaler$i"
            $w bind scaler$i <Button-1> "scaler_start h %W $i %x"
            $w bind scaler$i <B1-Motion> "scaler_move h %W $i %x"
        } else {
            $w create rect 0 [expr ($m+0.5)*$grid] \
                    $width [expr ($m+1.5)*$grid] \
                    -fill DarkGray -outline black -tag "scaler$i"
            $w bind scaler$i <Button-1> "scaler_start v %W $i %y"
            $w bind scaler$i <B1-Motion> "scaler_move v %W $i %y"
        }
    }
    set m 0
    for {set i 0} {$i < $domains} {incr i} {
        set n $scaler($w,$i,val)
        if {$dir == "h"} {
            $w create text [expr ($m+1)*$grid+$n*$grid/2] [expr $width/2] \
                    -text $n -tag "text$i"
        } else {
```

```
            $w create text [expr $width/2] [expr ($m+1)*$grid+$n*$grid/2] \
                    -text $n -tag "text$i"
        }
        incr m $n
    }
}


proc scaler_start {dir w i p} {
    global scaler
    if {$dir == "h"} {
        set p [$w canvasx $p $scaler($w,grid)]
    } else {
        set p [$w canvasy $p $scaler($w,grid)]
    }
    set scaler($w,$i,p) $p
}


proc scaler_move {dir w i p} {
    global scaler
    set grid $scaler($w,grid)
    if {$dir == "h"} {
        set p [$w canvasx $p $grid]
    } else {
        set p [$w canvasy $p $grid]
    }
    set p1 $scaler($w,$i,p)

    if {$p == $p1} return

    set scaler($w,$i,p) $p
    set m [expr $p/$grid-1]
    set d [expr $m-$scaler($w,$i,pos)]
    set i1 [expr $i+1]

    if {$scaler($w,$i,val)+$d < $scaler($w,$i,min)} return
    if {$scaler($w,$i,val)+$d > $scaler($w,$i,max)} return
    if {$scaler($w,$i1,val)-$d < $scaler($w,$i1,min)} return
    if {$scaler($w,$i1,val)-$d > $scaler($w,$i1,max)} return

    set var $scaler($w,var)
    global $var
    set scaler($w,$i,val) [expr int($scaler($w,$i,val)+$d)]
    $w itemconfigure text$i -text $scaler($w,$i,val)
    set ${var}($i) $scaler($w,$i,val)

    set scaler($w,$i,pos) [expr $scaler($w,$i,pos)+$d]

    set scaler($w,$i1,val) [expr int($scaler($w,$i1,val)-$d)]
    $w itemconfigure text$i1 -text $scaler($w,$i1,val)
    set ${var}($i1) $scaler($w,$i1,val)

    if {$dir == "h"} {
        $w move text$i [expr $d*$grid/2] 0
        $w move scaler$i [expr $d*$grid] 0
        $w move text$i1 [expr $d*$grid/2] 0
    } else {
        $w move text$i 0 [expr $d*$grid/2]
        $w move scaler$i 0 [expr $d*$grid]
        $w move text$i1 0 [expr $d*$grid/2]
    }
}

proc scaler_test {} {
    proc ende {} {
        global v1 v2 v3 h1 h2 h3
        puts "$v1(0) $v1(1) $v1(2)"
        puts "$v2(0) $v2(1) $v2(2)"
        puts "$v3(0) $v3(1) $v3(2) $v3(3) $v3(4)"
        puts "$h1(0) $h1(1) $h1(2)"
        puts "$h2(0) $h2(1) $h2(2)"
        puts "$h3(0) $h3(1) $h3(2) $h3(3) $h3(4)"
        exit
    }

    frame .links
    frame .rechts
    button .exit -text Exit -command ende
    scaler .v1 vert 20 20 30 v1 {3 3 3}
    scaler .v2 vert 10 20 30 v2 {{0 0 3} {2 3 4} {0 1 10}}
    scaler .v3 vert 30 20 30 v3 {0 5 -2 0 0}
    scaler .h1 hor 20 15 30 h1 {3 3 3}
    scaler .h2 hor 10 15 30 h2 {{0 0 3} {2 3 4} {0 1 10}}
    scaler .h3 hor 30 15 30 h3 {0 5 -2 0 0}
    pack .v1 .v2 .v3 -in .links -side left -anchor s
    pack .h1 .h2 .h3 -in .rechts -side top -anchor w
```

```
    pack .exit -in .rechts -side top -fill both -expand true
    pack .links .rechts -side left
}



# scale_minmax: ein scaler mit zwei Reglern, die einen Min- und
# einen Max-Wert angeben

proc scale_minmax {w dir len grid width var {doms {}}} {
    global scale_minmax
    global $var

# damit sind Abkuerzungen moeglich: vert..., vert, ver, ve, v,
# hor..., hor, ho, h
    set dir [string index $dir 0]

    set scale_minmax($w,len) $len
    set scale_minmax($w,grid) $grid
    set scale_minmax($w,width) $width
    if {[llength $doms] == 0} {
        set scale_minmax($w,min,min) 0
        set scale_minmax($w,min,val) 0
        set scale_minmax($w,min,max) $len
        set scale_minmax($w,max,min) 0
        set scale_minmax($w,max,val) $len
        set scale_minmax($w,max,max) $len
    } else {
        set scale_minmax($w,min,min) [lindex [lindex $doms 0] 0]
        set scale_minmax($w,min,val) [lindex [lindex $doms 0] 1]
        set scale_minmax($w,min,max) [lindex [lindex $doms 0] 2]
        set scale_minmax($w,max,min) [lindex [lindex $doms 1] 0]
        set scale_minmax($w,max,val) [lindex [lindex $doms 1] 1]
        set scale_minmax($w,max,max) [lindex [lindex $doms 1] 2]
    }
    set ${var}(min) $scale_minmax($w,min,val)
    set ${var}(max) $scale_minmax($w,max,val)
    set scale_minmax($w,var) $var

    if {$dir == "h"} {
        canvas $w -width [expr ($len+2)*$grid] -height $width
        $w create rect [expr $grid/2] 0 [expr ($len+1.5)*$grid] $width \
                -outline gray -fill white
    } else {
        canvas $w -width $width -height [expr ($len+2)*$grid]
        $w create rect 0 [expr $grid/2] $width [expr ($len+1.5)*$grid] \
                -outline gray -fill white
    }
    # die Farbe #f7f754543131 heisst 'tomato'
    foreach i {min max} c {green #f7f754543131} {
        set m $scale_minmax($w,$i,val)
        if {$dir == "h"} {
            $w create rect [expr ($m+0.5)*$grid] 0 \
                    [expr ($m+1.5)*$grid] $width \
                    -fill $c -outline black -tag "scale_minmax$i"
            $w create text [expr ($m+1)*$grid] [expr $width/2] \
                    -text $m -tag "text$i scale_minmax$i"
            $w bind scale_minmax$i <Button-1> "scale_minmax_start h %W $i %x"
            $w bind scale_minmax$i <B1-Motion> "scale_minmax_move h %W $i %x"
        } else {
            $w create rect 0 [expr (($len+2)-($m+0.5))*$grid] \
                    $width [expr (($len+2)-($m+1.5))*$grid] \
                    -fill $c -outline black -tag "scale_minmax$i"
            $w create text [expr $width/2] [expr (($len+2)-($m+1))*$grid] \
                    -text $m -tag "text$i scale_minmax$i"
            $w bind scale_minmax$i <Button-1> "scale_minmax_start v %W $i %y"
            $w bind scale_minmax$i <B1-Motion> "scale_minmax_move v %W $i %y"
        }
    }
}


proc scale_minmax_start {dir w i p} {
    global scale_minmax
    if {$dir == "h"} {
        set p [$w canvasx $p $scale_minmax($w,grid)]
    } else {
        set p [$w canvasy $p $scale_minmax($w,grid)]
    }
    set scale_minmax($w,$i,p) $p
}


proc scale_minmax_move {dir w i p} {
    global scale_minmax
    set grid $scale_minmax($w,grid)
    if {$dir == "h"} {
        set p [$w canvasx $p $grid]
```

```
    } else {
        set p [$w canvasy $p $grid]
    }
    set p1 $scale_minmax($w,$i,p)

    if {$p == $p1} return

    set scale_minmax($w,$i,p) $p
    if {$dir == "h"} {
        set m [expr $p/$grid-1]
    } else {
        set m [expr $scale_minmax($w,len)-$p/$grid+1]
    }
    set d [expr $m-$scale_minmax($w,$i,val)]

    if {$scale_minmax($w,$i,val)+$d < $scale_minmax($w,$i,min)} return
    if {$scale_minmax($w,$i,val)+$d > $scale_minmax($w,$i,max)} return
    if {$i == "min"} {
        if {$scale_minmax($w,min,val)+$d > $scale_minmax($w,max,val)} return
    } else {
        if {$scale_minmax($w,min,val) > $d+$scale_minmax($w,max,val)} return
    }

    set var $scale_minmax($w,var)
    global $var

    set scale_minmax($w,$i,val) [expr int($scale_minmax($w,$i,val)+$d)]
    $w itemconfigure text$i -text $scale_minmax($w,$i,val)
    set ${var}($i) $scale_minmax($w,$i,val)

    if {$dir == "h"} {
        $w move scale_minmax$i [expr $d*$grid] 0
    } else {
        $w move scale_minmax$i 0 [expr -$d*$grid]
    }
}


proc scale_minmax_test {} {
    proc ende {} {
        global v1 v2 v3 h1 h2 h3
        puts "$v1(min) $v1(max)"
        puts "$v2(min) $v2(max)"
        puts "$v3(min) $v3(max)"
        puts "$h1(min) $h1(max)"
        puts "$h2(min) $h2(max)"
        puts "$h3(min) $h3(max)"
        exit
    }

    frame .links
    frame .rechts
    button .exit -text Exit -command ende
    scale_minmax .v1 vert 20 20 30 v1
    scale_minmax .v2 vert 10 20 30 v2 {{0 3 5} {3 10 10}}
    scale_minmax .v3 vert 30 20 30 v3
    scale_minmax .h1 hor 20 20 30 h1
    scale_minmax .h2 hor 10 20 30 h2 {{0 3 5} {3 10 10}}
    scale_minmax .h3 hor 30 20 30 h3
    pack .v1 .v2 .v3 -in .links -side left -anchor s
    pack .h1 .h2 .h3 -in .rechts -side top -anchor w
    pack .exit -in .rechts -side top -fill both -expand true
    pack .links .rechts -side left
}
```

A.14 bitmaps.tcl

`bitmaps.tcl` lädt zusätzliche Bitmap-Dateien vor allem zur schöneren Darstellung von „Knöpfen“.

```
foreach f [glob std/bitmaps/*] {
    set i [file rootname [lindex [file split $f] end]]
    image create bitmap $i -file $f
}
```

A.15 place-window.tcl

`place-window.tcl` definiert eine Prozedur zur Platzierung eines Tk-Fensters, zum Beispiel in der Bildschirm-Mitte. Der Prolog beschreibt die Herkunft (den Urheber) der Prozedur.

```
# From: jhobbs@cs.uoregon.edu (Jeffrey Hobbs)
# Newsgroups: comp.lang.tcl
# Subject: Re: Centering of Widgets
# Date: 21 Mar 1997 17:18:53 -0800
# Organization: University of Oregon Computer and Information Sciences Dept.
# Lines: 49
# Message-ID: <5gvc1t$5ce@psychotix.cs.uoregon.edu>
# References: <3332F615.59E1@ssd.bna.boeing.com>
# NNTP-Posting-Host: psychotix.cs.uoregon.edu

# In article <3332F615.59E1@ssd.bna.boeing.com>,
# Doug Grandey  <dsgrande@ssd.bna.boeing.com> wrote:
# >screen.  Is there any way to place the entire toplevel in the relative
# >center of the user's screen?

# Use the following proc:



################################################################
## place_window: place a toplevel at a particular position
##
## ARGS: toplevel ?placement_type? ?args?
## toplevel     - name of toplevel window
## placement    - pointer ?center? ; places $w centered on the pointer
##                widget widget_name ; centers $w over widget_name
##                defaults to placing toplevel in the middle of the screen
##
proc place_window {w {p ""} {a ""}} {
  wm withdraw $w
  update idletasks
  switch -glob -- $p {
    p* {
      ## place at POINTER (centered is $a == center)
      if [string match c* $a] {
        wm geometry $w +[expr \
            [winfo pointerx $w]-[winfo width $w]/2]+[expr \
            [winfo pointery $w]-[winfo height $w]/2]
      } else {
        wm geometry $w +[winfo pointerx $w]+[winfo pointery $w]
      }
    }
    w* {
      ## center about WIDGET $a
      wm geometry $w +[expr [winfo rootx $a]+([winfo width $a]\
          -[winfo width $w])/2]+[expr \
          [winfo rooty $a]+([winfo height $a]-[winfo height $w])/2]
    }
    default {
      wm geometry $w +[expr ([winfo screenwidth $w]-[winfo reqwidth $w]) \
          / 2]+[expr ([winfo screenheight $w]-[winfo reqheight $w]) / 2]
    }
  }
  wm deiconify $w
}


# --
#   Jeffrey Hobbs                          jhobbs@cs.uoregon.edu,@or.cadix.com
#   Software Engineer, Oregon R&D          office: 541.683.7891
#   CADIX International, Inc.              fax:    541.683.8325
#              URL: http://www.cs.uoregon.edu/~jhobbs/
```

A.16 interdip-defaults

Diese Datei legt (als X-Ressource) im Moment nur die Schriftart der Anwendung fest. Damit kann im Prinzip die ganze Erscheinung von INTERDIP verändert werden.

```
!*font: *courier-bold-r*14*
!*font: *helvetica-medium-r*14*
!*font: *helvetica-medium-r*18*
!*font: *lucida-bold-r*12*
*font: *lucida-medium-r*12*
```

Literaturverzeichnis

[...]

[aH96] Harald Meyer auf'm Hofe. Representation of Requirements Through Preference Orderings of Soft Constraints. Technical report, DFKI Kaiserslautern, Feb. 1996. http://www.dfki.uni-kl.de/conplan/Welcome.html.

[aHT95] Harald Meyer auf'm Hofe und Bidjan Tschaitschian. PCSPs with Hierarchical Constraint Orderings in Real World Scheduling Applications. In *Notes on the CP'95 Workshop on Over-Constrained Systems*, Seiten 69–76, Cassis, France, 1995.

[AS97] Slim Abdennadher und Hans Schlenker. INTERDIP – Ein interaktiver Constraint-basierter Dienstplaner für Krankenstationen. In *Proceedings 12. Workshop logische Programmierung – WLP'97*, 1997.

[Bar95] Victor A. Bardadym. Computer-Aided School And University Timetabling: The New Wave. In *ICPTAT '95 – Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling*, Seiten 253–268, 1995.

[BMMW89] Borning, Maher, Martindale und Wilson. Constraint Hierarchies and Logic Programming. 1989.

[Con] ConPlan. http://www.dfki.uni-kl.de/conplan/Welcome.html.

[CS93] Volker Claus und Andreas Schwill. *Duden Informatik*. Dudenverlag, Mannheim, Leipig, Wien, Zürich, 1993.

[DFM+94] S. J. Darmoni, A. Fajner, N. Mahe, A. Leforestier, M. Vondracek, O. Stelian und M. Baldenweck. HOROPLAN: Computer-Assisted Nurse Scheduling Using Constraint-Based Programming. *Journal of the Society for Health Systems*, 1994.

[DVS+88] Mehmet Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf und F. Berthier. The Constraint Logic Programming Language CHIP. Technical Report TR-LP-37, ECRC, Munich, Germany, Mai 1988.

[FA97] Thom Frühwirth und Slim Abdennadher. *Constraint-Programmierung: Grundlagen und Anwendungen*. Springer, 1997.

[FHK+93] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry le Provost, Pierre Lim, Eric Monfroy und Mark Wallace. Constraint Logic Programming – An Informal Introduction. Technical report, ECRC, 93.

[FM94] E. C. Freuder und A. K. Mackworth. *Constraint-Based Reasoning.* MIT Press, Cambridge, Massachusetts, 1994.

[Fre91] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *AAAI-91 – Proceedings of the 9th national conference on artificial intelligence*, Seiten 227–233, 1991.

[Frü95] T. Frühwirth. Constraint Handling Rules. In A. Podelski, Hrsg., *Constraint Programming: Basics and Trends*, LNCS 910, Berlin, März 1995. Springer-Verlag.

[FW92] E. C. Freuder und R. J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58, 1992.

[HW96] Kamel Heus und Georges Weil. Constraint Programming A Nurse Scheduling Application. In *PACT 96 – Proceedings of the Second International Conference on the Practical Application of Constraint Technology*, Seiten 115–127. The Practical Application Company, 1996.

[ISO94] Draft International Standard, ISO/IEC DIS 13211-1, 1994.

[JM94] J. Jaffar und M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 20:503–581, 1994.

[Kre91] Ulrich Krengel. *Einführung in die Wahrscheinlichkeitstheorie und Statistik.* Vieweg, Braunschweig, 1991.

[Kum92] Vipin Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 1992.

[LA96] J. M. Lazaro und P. Aristondo. Job Rostering with Constraints. In *PACT 96 – Proceedings of the Second International Conference on the Practical Application of Constraint Technology*, Seiten 155–168. The Practical Application Company, 1996.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[Sch92] Thomas Schiex. Possibilistic Constraint Satisfaction or "How to Handle Soft Constraints". In *Proccedings of the Eighth Conference of Uncertainty in Artificial Intelligence*, 1992.

[Sch95] Wolfgang Schelter. *Das Tarifrecht der Angestellten im Pflegedienst (BAT).* Courier Verlag, Stuttgart, 1995.

[Sie96a] Siemens Nixdorf Informationssysteme AG. *IF/Prolog Constraint Problem Solver*, 1996.

[Sie96b] Siemens Nixdorf Informationssysteme AG. *IF/Prolog Users Guide*, 1996.

[SS94] Leon Sterling und Ehud Shapiro. *The art of Prolog*. MIT Press, Cambridge, Massachusetts, 2. Auflage, 1994.

[Van89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[Van91] Pascal Van Hentenryck. Constraint Logic Programming. *The Knowledge Engineering Review*, 6:151–194, 1991.

[vdB93] Bart van den Bosch. Implementation of a CLP library and an application in nurse scheduling. Diplomarbeit, Katholieke Universiteit Leuven, Belgium, 1993.

[VSD93] Pascal Van Hentenryck, Vijay Saraswat und Yves Deville. Design, Implementations and Evaluation of the Constraint Language cc(FD). Technical Report CS-93-02, Brown University, Providence, January 1993.

[Wel95] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1995.

[WH95] Georges Weil und Kamel Heus. Eliminating Interchangeable Values in the Nurse Scheduling Problem Formulated as a Constraint Satisfaction Problem. In *CONSTRAINT'95*, 1995.

[WHG96] Weißschnur, Hertzberg und Guesgen. In Search of an Efficient Technique for Constraint Relaxation. In *Proccedings of the Constraint-96 Workshop*, 1996.

Index