Problemlösen mit constraintbasierter Programmierung

Hans-Joachim Goltz

April 1999

GMD – Forschungszentrum Informationstechnik GmbH

Forschungsinstitut für Rechnerarchitektur und Softwaretechnik

GMD-FIRST, Kekulèstr. 7, D-12489 Berlin
Tel.: (030) 63921869 Fax: (030) 63921805 e-mail: goltz@first.gmd.de

Zusammenfassung

Das Konzept der constraintbasierten Programmierung wird in diesem Bericht kurz vorgestellt und an Beispielen werden Arbeitsweise, Suchmethoden und Modellierung eines Planungsproblems erläutert. Das Programmierparadigma der constraintbasierten Programmierung unterscheidet sich wesentlich von dem der klassischen Programmierung. Durch den deklarativen Charakter der Constraints ist eine effiziente und flexible Softwareentwicklung möglich und die Entwicklung korrekter Software wird unterstützt. Die Art und Weise der Verarbeitung bzw. der Berücksichtigung der Constraints führt zu großen Suchraumeinschränkungen und ermöglicht die schnelle Erzeugung von Lösungen, die nahe dem Optimum liegen. Die constraintbasierte Programmierung kann insbesondere zum Lösen komplexer Suchprobleme, die mittels Constraints (Bedingungen) formulierbar sind, erfolgreich eingesetzt werden. Die Anzahl der industriellen Anwendungen der constraintbasierten Programmierung wächst zur Zeit sehr schnell.

1 Einleitung

Die Anzahl der industriellen Anwendungen der constraintbasierten Programmierung wächst zur Zeit sehr schnell. Auch die internationalen Forschungsaktivitäten sind in diesem Gebiet in den letzten Jahren ständig gestiegen. Jährliche internationale Konferenzen zur constraintbasierten Programmierung und zur praktischen Anwendung constraintbasierter Technologien gibt es seit 1995. Das Programmierparadigma der constraintbasierten Programmierung ist noch relativ neu. Obwohl bereits seit Anfang der 60'er Jahre Forschungen zu constraintbasierten Methoden durchgeführt werden, wurde dieses Paradigma erstmalig 1987 als Paradigma der constraintlogischen Programmierung vorgestellt und auch theoretisch begründet. Dieses Konzept der constraintlogischen Programmierung hatte einen sehr großen Einfluß auf die Entwicklung von constraintbasierten Programmiersprachen.

Das Programmierparadigma der constraintbasierten Programmierung unterscheidet sich wesentlich von dem der klassischen Programmierung. Durch den deklarativen Charakter der Constraints ist eine effiziente und flexible Softwareentwicklung möglich und die Entwicklung korrekter Software wird unterstützt. Problemmodifikationen sind relativ einfach zu realisieren. Die Art und Weise der Verarbeitung bzw. der Berücksichtigung der Constraints führt zu großen Suchraumeinschränkungen und ermöglicht die schnelle Erzeugung von Lösungen, die nahe dem Optimum liegen. Die constraintbasierte Programmierung kann insbesondere zum Lösen komplexer Suchprobleme, die mittels Constraints (Bedingungen) formulierbar sind, erfolgreich eingesetzt werden. Anwendungsgebiete sind beispielsweise Probleme aus dem Bereich Produktionsplanung, wie Ablaufplanung, Kapazitätsplanung, Transportplanung und Personalplanung, und aus dem Bereich Konfiguration, wie Schaltkreisentwurf, Konfiguration von Produkten und Konfiguration technischer Anlagen.

Das Konzept der constraintbasierten Programmierung wird in diesem Bericht kurz vorgestellt und an Beispielen werden Arbeitsweise, Suchmethode und Modellierung eines Planungsproblems diskutiert. Im nächsten Abschnitt werden die Begriffe *Constraint* und *Constraint-Löser* erläutert. Im 3. Abschnitt wird gezeigt, wie einfach die wichtigsten Berechnungen, die in der Netzplantechnik erforderlich sind, bei der Anwendung constraintbasierter Methoden durchgeführt werden können. Als Beispiel einer constraintbasierten Programmiersprache wird im 4. Abschnitt das Konzept der constraintlogischen Programmierung vorgestellt, wobei insbesondere auf die Arbeitsweise des Constraint-Lösers eingegangen wird. An einem kleinen Beispiel wird im 5. Abschnitt die Modellierung und das Lösen eines Problems der Ablaufplanung in einer constraintlogischen Programmiersprache diskutiert. Insbesondere wird in diesem Abschnitt deutlich, wie verschiedenartige Bedingungen in dieser Programmiersprache repräsentiert werden können. Im 6. Abschnitt werden an einem sehr einfachen Planungsbeispiel verschiedene Suchmethoden beschrieben und verglichen.

2 Constraints und Constraint-Löser

Constraints dienen zur Repräsentation von relationalen Beziehungen zwischen Variablen. Durch Constraints wird ausgedrückt, welche Bedingungen zulässige Belegungen der Variablen erfüllen müssen. Eine Lösung einer Menge von Constraints ist folglich eine Belegung der in dieser Menge vorkommenden Variablen, so daß alle diese Constraints erfüllt sind. Lineare Gleichungen und Ungleichungen sind beispielsweise Constraints. Weitere Beispiele werden in den folgenden Ausführungen genannt. Eine Vielzahl von Problemen kann mittels Constraints formuliert werden. Da Constraints ungerichtete Zusammenhänge zwischen Variablen repräsentieren, wird keine konkrete Problemlösung festgelegt. Es werden nur Bedingungen gestellt, die eine Problemlösung erfüllen muß.

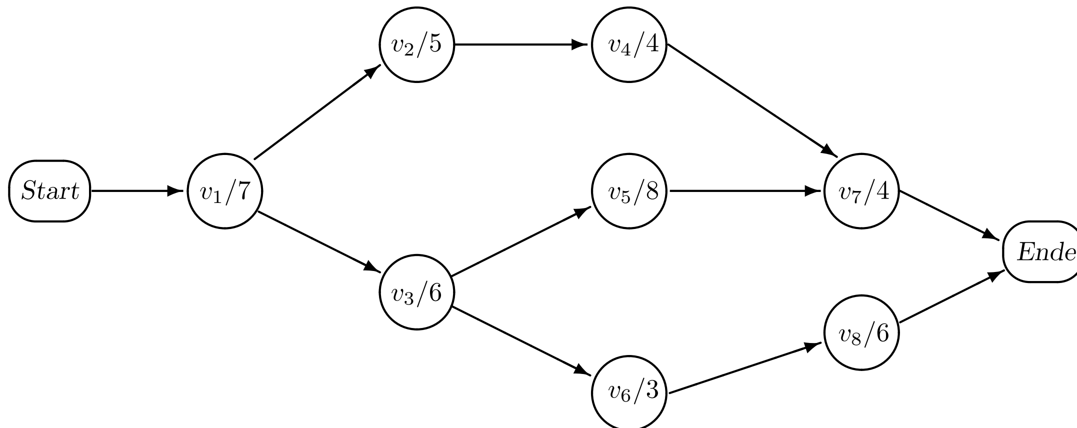
Die Verarbeitung von Constraints erfolgt durch einen *Constraint-Löser*. Seine Arbeitsweise besteht im Vereinfachen von Constraintmengen, in Konsistenzprüfungen und im Erzeugen von Konsequenzen und allgemeinen Lösungen, die beim Festlegen aller Variablenwerte auch konkret sein können. Ein Constraint-Löser grenzt im allgemeinen die Menge der potentiell möglichen Lösungen ein, falls er nicht Widersprüche erkennt. Um konkrete Lösungen zu finden, sind oft in einem zusätzlichen Schritt noch weitere Constraints hinzuzufügen oder einem Teil der Variablen schrittweise Werte zuzuordnen. Obwohl die Bezeichnung *Constraint-Löser* das Ziel der Constraintverarbeitung nicht richtig charakterisiert, werden wir diese Bezeichnung hier verwenden, da sie sehr gebräuchlich ist. Verfahren zur Behandlung von Constraints sind schon lange ein Gegenstand der KI-Forschung (Constraint Satisfaction Problem). Constraint-Löser enthalten oft Methoden aus verschiedenen Gebieten: Mathematik, Operations Research, Künstliche Intelligenz, Geometrie.

Constraint-Löser unterscheiden sich bezüglich der Wertebereiche (Domänen) für die Variablen und bezüglich Semantik und Syntax der Ausdrücke, mit denen Constraints formuliert werden können. Es existieren Constraint-Löser für Gleichungen zwischen Booleschen Termen, für lineare Gleichungen und Ungleichungen über rationale bzw. reelle Zahlen, und für verschiedenartige Relationen über endlichen Domänen, insbesondere über endlichen Mengen natürlicher Zahlen.

Für die Arbeitsweise eines Constraint-Lösers ist die inkrementelle Verarbeitung der Constraints wichtig. Beim Hinzufügen von Constraints zu einer bestehenden Constraintmenge sollte der Constraint-Löser nicht eine komplette Neuverarbeitung beginnen, sondern nur die Schlußfolgerungen aus dem Hinzufügen berechnen.

3 Netzplantechnik mit constraintbasierten Methoden

Die Netzplantechnik wird für die Planung und Kontrolle von Produktionsprozessen und von Projektierungs- und Konstruktionsaufgaben verwendet. Sie wendet graphentheoretische Methoden auf speziellen Graphen, den Netzplänen, an. Ein Netzplan ist, mathematisch gesehen, ein kanten- oder/und knotenbewerteter gerichteter Graph ohne Kreise mit genau einer Quelle und genau einer Senke. Als Beispiel betrachten wir den folgenden Netzplan:



Jedem Knoten ist ein Vorgang (Arbeitsgang oder Teilaufgabe) mit einer Zeitdauer in einer gegebenen Zeiteinheit zugeordnet. Für den Startknoten und den Endeknoten wird dabei jeweils die Zeitdauer 0 vorausgesetzt. Die Kanten kennzeichnen die Reihenfolgebeziehungen zwischen den Vorgängen. In dem gegebenen Netzplan muß beispielsweise der Vorgang v_3 beendet sein, bevor die Vorgänge v_5 und v_6 beginnen können. Netzpläne können auch so definiert werden, daß die gerichteten Kanten mit Vorgängen und die Knoten mit Ereignissen bewertet werden. Wichtige Ziele der Anwendung der Netzplantechnik sind:

- Bestimmumg des frühesten Endtermins des Gesamtvorhabens,
- Bestimmung der frühesten und spätesten Startzeit jedes Vorganges,
- Bestimmung des kritischen Weges.

Wenn vorausgesetzt wird, daß der früheste Endtermin erreicht werden soll, ist ein kritischer Weg des Netzplanes ein Weg vom Startknoten zum Endeknoten, zu dem nur die Knoten von Vorgängen gehören, bei denen die früheste Startzeit gleich der spätesten Startzeit ist. Die Kenntnis des kritischen Weges in einem Netzplan ist wichtig, da die Vergrößerung der Dauer oder der verspätete Beginn auch nur eines Vorganges auf diesem Weg sofort zu einem späteren Endtermin des Gesamtvorhabens führt.

Mittels eines Constraint-Lösers für endliche Domänen können diese Berechnungen im Netzplan sehr leicht ausgeführt werden. Wenn wir voraussetzen, daß die Startzeit `Start` und der späteste Endtermin `MaxEnd` gegeben sind, dann sind zuerst die folgenden Schritte als Eingabe in einem Constraint-Löser zu realisieren:

Vereinbarung von Domänenvariablen :

Die Startzeiten der Vorgänge werden als Variablen betrachtet, die als Werte nur natürliche Zahlen aus dem Intervall { `Start..MaxEnd` } annehmen können.

Absetzen von Constraints :

Die Ungleichungen, die sich aus den direkten Reihenfolgebeziehungen der Vorgänge ergeben, werden als Constraints formuliert.

Der Constraint-Löser streicht dann aus den Domänen der Variablen jeweils alle Werte, die nicht zu einer Lösung gehören können. Die Reihenfolge der Eingabe der Ungleichungen in den Constraint-Löser hat keinen Einfluß auf das Ergebnis der Domänenreduzierungen. Der früheste Endtermin des Gesamtvorhabens ist der kleinste Wert der Variablendomäne, die dem Endtermin zugeordnet ist. Wenn als zusätzliche Eingabe in den Constraint-Löser, dieser Variable dieser kleinste Wert zugewiesen wird (d.h., früheste Endtermin ist gleich dem spätesten Endtermin), ergibt sich sofort:

- Der kleinste Wert einer Domäne ist die früheste Startzeit des zugehörigen Vorganges.
- Der größte Wert einer Domäne ist die späteste Startzeit des zugehörigen Vorganges.
- Alle Vorgänge, deren zugehörigen Variablendomänen einelementig sind, gehören zu einem kritischen Weg.

Wir betrachten nun den gegebenen Netzplan und setzen als Startzeit den Zeitpunkt 0 und als spätesten Endtermin den geschätzten Zeitpunkt 32 voraus. Weiterhin seien `V1, V2, ..., End` jeweils die Variablen für die möglichen Startzeiten der Vorgänge v_1, v_2, …, *Ende*. Zuerst sind die Vereinbarungen der Domänen dieser Variablen zu realisieren:

```
[ V1, V2, V3, V4, V5, V6, V7, V8, End ] in {0..32}
```

Aus dem gegebenen Netzplan ergeben sich die folgenden Ungleichungen aus den Reihenfolgebeziehungen der Vorgänge:

```
V1 + 7 =< V2,   V1 + 7 =< V3,   V2 + 5 =< V4,   V3 + 6 =< V5,
V3 + 6 =< V6,   V4 + 4 =< V7,   V5 + 8 =< V7,   V6 + 3 =< V8,
V7 + 4 =< End,  V8 + 6 =< End
```

Nach der Eingabe der genannten Domänenvereinbarungen und der Ungleichungen in den Constraint-Löser erhält man als Ausgabe:

```
V1 in {0..7}, V2 in {7..19}, V3 in {7..14}, V4 in {12..24},
V5 in {13..20}, V6 in {13..23}, V7 in {21..28},
V8 in {16..26}, End in {25..32}
```

Folglich ist der früheste Endtermin der Zeitpunkt 25. Wenn nun zusätzlich die Gleichung `End = 25` in den Constraint-Löser eingegeben wird, ergibt sich:

```
V1 = 0, V2 in {7..12}, V3 = 7, V4 in {12..17}, V5 = 13,
V6 in {13..16}, V7 = 21, V8 in {16..19}, End = 25
```

Der kritsche Weg des gegebenen Netzplans ist somit die Folge der Knoten: *Start*, v_1, v_3, v_5, v_7, *Ende*. Die frühesten und die spätesten Startzeiten der anderen Vorgänge ergeben sich direkt aus den Domänen der zugehörigen Variablen.

4 Constraintlogische Programmierung

Im allgemeinen ist eine constraintbasierte Programmiersprache eine Integration von Constraint-Lösern in eine Programmiersprache, so daß eine spezifische Verarbeitung der Constraints ermöglicht wird. Prinzipiell könnten Constraint-Löser in verschiedenartigen Programmiersprachen integriert werden. Durch den ILOG-Solver[1] wird beispielsweise eine constraintbasierte Programmierung mit C++ realisiert.
Die *constraintlogische Programmierung* ist eine Erweiterung des Programmierparadigmas der Logischen Programmierung um die effiziente Verarbeitung von Constraints und vereinigt somit die Deklarativität der Logischen Programmierung mit der Effizienz des Constraint-Lösens. Da Logische Programmiersprachen relationale Sprachen sind, ist dies eine natürliche Kombination, die auch mathematisch fundiert ist. Außerdem enthält die Logische Programmierung ein Inferenzsystem mit einem Backtracking-Mechanismus, das die Programmierung einer Lösungssuche wesentlich vereinfacht. In einer constraintlogischen Programmiersprache ist in dem Backtracking-Mechanismus auch das Zurücksetzen des Zustandes der Constraintverarbeitung enthalten.

[1] der ILOG-Solver ist ein Produkt der Firma ILOG aus Frankreich

Die grundlegende Arbeitsweise eines Constraint-Lösers innerhalb der constraintlogischen Programmierung kann wie folgt beschrieben werden:

- *Die Constraints werden soweit wie möglich berücksichtigt, wobei aber die Lösungsmenge nicht reduziert wird, d.h., der Constraint-Löser selbst erzeugt keine Lösungsauswahl. Die Constraints, die nicht bzw. nicht vollständig auswertbar sind, werden zurückgestellt. Ein Constraint wird nur dann nicht zurückgestellt, wenn alle möglichen Lösungen dieses Constraint erfüllen.*

- *Constraints, die auswertbar bzw. teilweise auswertbar sind, werden sofort bearbeitet und die Auswirkungen dieser Bearbeitung auf die Menge der zurückgestellten Constraints wird untersucht.*

- *Ein Constraint im Wartezustand wird wieder aktiviert, wenn neue Information, die dieses Constraint betrifft, hinzukommt. Insbesondere werden die Auswirkungen einer Wertzuweisung (auch außerhalb des Constraint-Lösers) auf die zurückgestellten Constraints untersucht.*

- *Wird bei dieser Bearbeitung eine Inkonsistenz festgestellt, wird Backtracking eingeleitet, d.h., es wird auf einen Zustand des Systems zurückgesetzt, in dem noch eine andere Wahl der Programmfortsetzung existierte.*

Im gewissen Sinne arbeitet ein Constraint-Löser im "Hintergrund". Er wacht darüber, daß die Lösungen die Constraints erfüllen, wobei er soweit wie es ihm möglich ist, den Suchraum reduziert und Constraintmengen vereinfacht. Ein wichtiger Bestandteil der Arbeitsweise ist dabei der sogenannte "*Delay-Mechanismus*", durch den die verzögerte Constraintauswertung ermöglicht wird.

An einem kleinen Beispiel wird im folgenden die Arbeitsweise eines Constraint-Lösers innerhalb einer constraintlogischen Programmiersprache erläutert. Vorausgesetzt wird, daß Constraints über Variablen mit endlichen Domänen verarbeitet werden können und daß u.a. Ungleichungen Constraints sind. Wir betrachten die sequentielle Bearbeitung der drei Aufrufe:

```
[X,Y,Z] in {1..20},  X + 5 < Y,  Z + 10 < X
```

Durch den ersten Aufruf werden den Variablen `X, Y, Z` jeweils die natürlichen Zahlen `1` bis `20` als Domäne zugeordnet. Der anschließende Aufruf des Constraints `X + 5 < Y` aktiviert den Constraint-Löser, der feststellt, daß `X` kleiner als 15 und `Y` größer als 6 sein muß.

Entsprechend werden die Domänen dieser beiden Variablen eingeschränkt. Damit ist dieses Constraint aber noch nicht gelöst, da trotz der Domäneneinschränkung Wertezuordnungen existieren, die dieses Constraint nicht erfüllen. Deshalb geht es in einen Wartezustand und "wacht" über seine Erfüllung.

Aus dem zweiten Constraint `Z + 10 < X` folgert der Constraint-Löser dann, daß `X` nur die Werte 12, 13 oder 14 annehmen kann und `Z` kleiner als 4 sein muß. Da nun die Domäne von `X` weiter eingeschränkt wurde, wird das erste Constraint wieder aufgeweckt und die Domäne von `Y` wird auf {18, 19, 20} eingeschränkt. Beide Constraints gehen in den Wartezustand und würden bei weiteren Informationen wieder aktiviert werden. Als Ergebnis bzw. Zwischenergebnis erhalten wir also:

```
X in {12..14},  Y in {18..20}  Z in {1..3}
```

Infolge der weiteren Verarbeitung könnten dann die 10 Wertetripel erzeugt werden, die die gegebenen Constraints erfüllen.

Beispiele constraintlogischer Programmiersprachen sind[2]: CHIP, CLP($\mathcal{R}$), PROLOG III, ECL^{i}PS^e. Die constraintlogische Programmierung konnte bereits zum Lösen verschiedener praktischer komplexer Suchprobleme erfolgreich eingesetzt werden. Vorteile der constraintlogischen Programmierung sind unter anderem:

- *Die Zuordnung eines zulässigen Wertes zu einer Constraintvariablen führt unmittelbar zu einer Suchraumeinschränkung. Eine unzulässige Wertzuordnung wird unmittelbar erkannt und zurückgewiesen.*
- *Durch Constraints wird festgelegt,* was *zu erfüllen ist; es wird nicht festgelegt,* wie *es zu erfüllen ist. Folglich sind Constraints deklarativ und die Modellierung von Problemen ist weitgehend unabhängig von den Lösungsmethoden, wodurch eine größere Flexibilität bei Änderungen der Problemstellung gewährleistet ist.*

Für die Repräsentation komplexe Zusammenhänge hat sich die Integration verschiedenartiger Methoden zu einem Verfahren der Constraintbehandlung als sehr vorteilhaft erwiesen. Die Nutzung dieser oft umfangreichen Programmpakete erfolgt auf hohem Abstraktionsniveau durch die Angabe von relationalen Beziehungen zwischen Objekten, die Variablen enthalten

[2] CHIP ist ein Produkt der Firma COSYTEC, Frankreich; CLP($\mathcal{R}$) ist ein Forschungsprototyp des IBM T. J. Watson Reserch Center, Yorktown Heigths, USA; PROLOG III ist ein Produkt der Firma PrologIA, Frankreich; ECL^{i}PS^e ist ein Forschungsprototyp der ECRC GmbH, München

können. Derartige Konstrukte werden als *globale Constraints* bezeichnet. In CHIP existiert beispielsweise ein Constraint "`cycle`" mit dem Zyklen in einem gerichteten Graphen erzeugt werden können. Dieses Constraint kann sehr gut zum Lösen von praktischen Problemen der Transportplanung eingesetzt werden, wobei auch unterschiedliche Kosten berücksichtigt werden können.

Ein weiteres globales Constraint ist das Constraint "`cumulative`", das zuerst in CHIP integriert wurde und inzwischen in verschiedenen constraintbasierten Programmiersprachen zur Verfügung steht. Durch dieses Constraint kann die Beschränkung der gleichzeitigen Ausführung von Arbeitsaufgaben durch eine vorhandene maximale Ressourcenverfügbarkeit ausgedrückt werden. Seien bespielsweise $T_1, T_2, T_3, \ldots, T_n$ Arbeitsaufgaben, die die gleiche nicht konsumierbare Ressource benötigen, und sei L die maximal zur Verfügung stehenden Anzahl dieser Ressource. Für jede Arbeitsaufgabe T_i sei weiterhin S_i die Startzeit, D_i die Dauer und H_i die benötigte Anzahl der Ressource. Zu jedem Zeitpunkt dürfen nicht mehr als $L-$viele Ressourcen angefordert werden. Mathematisch kann dies wie folgt ausgedrückt werden:

$$\text{für jedes} \quad k \in [\,min\{S_i\}\,,\; max\{S_i + D_i\} - 1\,] \quad \text{gilt:}$$
$$\Sigma\, H_j \;\leq\; L \quad \text{für alle} \;\; j \quad \text{mit} \quad S_j \leq k \leq S_j + D_j - 1$$

Durch das Constraint "`cumulative`" kann diese komplexe Bedingung wie folgt modelliert werden:

```
cumulative([S1,S2,...,Sn],[D1,D2,...,Dn],[H1,H2,...,Hn],L)
```

5 Modellierung und Lösen eines Planungsproblems

Die Modellierung und das Lösen eines Problems der Ablaufplanung mittels der constraintlogischen Programmierung wird in diesem Abschnitt an einem kleinen Beispiel erläutert. Wir betrachten die Planung zweier Aufträge (`a` und `b`), die jeweils aus fünf Arbeitsaufgaben bestehen. Für die Erfüllung einer Arbeitsaufgabe wird jeweils eine Maschine und Personal benötigt. Dieser Ressourcenbedarf und die Bearbeitungsdauer (in einer namenlosen Zeiteinheit) sind in der folgenden Tabelle gegeben.

Arbeits-aufgabe	Bearbeitungs-dauer	Nr. der Maschine	Anzahl des benötigten Personals
a1	4 oder 6	1	2 oder 1
a2	4	3	2
a3	3	2	1
a4	5	3	3
a5	4	1	1
b1	5	1	2
b2	4	2	1
b3	6	1	2
b4	4	2	3
b5	3	3	2

Außerdem sind die folgenden Bedingungen (Constraints) zu erfüllen:

1. *Die Bearbeitungsdauer der Aufgabe* `a1` *hängt von dem zur Verfügung stehendem Personal ab und beträgt* `4` *Zeiteinheiten bei* `2` *Personen und* `6` *Zeiteinheiten bei einer Person.*
2. *Zum Zeitpunkt* `15` *darf auf der Maschine* `2` *keine Aufgabe bearbeitet werden. Es dürfen zu diesem Zeitpunkt aber Arbeitsaufgaben beendet bzw. begonnen werden.*
3. *Die Bearbeitung der beiden Aufträge kann zum Zeitpunkt* `0` *beginnen und muß zum Zeitpunkt* `27` *abgeschlossen sein.*
4. *Zwischen den Arbeitsaufgaben des Auftrages* `a` *müssen folgende Bedingungen erfüllt sein:*
   (a) *Die Arbeitsaufgabe* `a2` *kann frühestens* `1` *Zeiteinheit vor und muß spätestens* `3` *Zeiteinheiten nach Beendigung der Aufgabe* `a1` *beginnen.*
   (b) *Die Arbeitsaufgabe* `a3` *kann beginnen: frühestens* `1` *Zeiteinheit und spätestens* `3` *Zeiteinheiten nach Beendigung der Aufgabe* `a2`*.*
   (c) *Die Arbeitsaufgabe* `a4` *kann beginnen: frühestens direkt und spätestens* `3` *Zeiteinheiten nach Beendigung der Aufgabe* `a2`*.*
   (d) *Die Arbeitsaufgabe* `a5` *kann fühestens* `3` *Zeiteinheiten nach Beendigung der Aufgaben* `a3` *und* `a4` *beginnen.*
5. *Die Arbeitsaufgaben des Auftrages* `b` *sind in der gegebenen Reihenfolge (* `b1,...,b5` *) zu bearbeiten, wobei eine Arbeitsaufgabe erst dann beginnen kann, wenn die vorangehende Aufgabe beendet ist.*
6. *Zu keinem Zeitpunkt darf auf einer Maschine mehr als eine Arbeitsaufgabe bearbeitet werden.*
7. *Insgesamt stehen* `4` *Personen für die Erfüllung dieser beiden Aufträge zur Verfügung. Folglich dürfen zu keinem Zeitpunkt mehr als* `4` *Personen für die Bearbeitung der Aufgaben erforderlich sein.*

Das Problem ist gelöst, wenn jeder Arbeitsaufgabe eine Startzeit und zusätzlich der Arbeitsaufgabe `a1` die Anzahl des zur Verfügung stehenden Personals (1 oder 2) zugeordnet sind, so daß alle Constraints erfüllt sind. Zum Lösen dieses Problems wird die constraintlogische

Programmierung mit einem Constraint-Löser über endlichen Domänen natürlicher Zahlen genutzt.

Seien `A1, A2, A3, A4, A5, B1, B2, B3, B4, B5` die Variablen für die Startzeiten der Arbeitsaufgaben, `Pa1` und `Da1` Variablen für Personal bzw. Dauer der Aufgabe `a1`. Das folgende Programm ist eine Modellierung dieses Problems, wobei die Modellierung der oben formulierten Constraints durch die Angabe der entsprechenden Nummer gekennzeichnet wird.

```
StartTasks = [A1,A2,A3,A4,A5,B1,B2,B3,B4,B5],

StartTasks in {0..27},     Pa1 in {1,2},

element(Pa1,[6,4],Da1),                                        (1)

notin(A3,13,14), notin(B2,12,14), notin(B4,12,14),             (2)

A5 + 4 =< 27, B5 + 3 =< 27,                                    (3)

A1 + Da1 =< A2 + 1, A2 =< A1 + Da1 + 3,                        (4a)

A2 + 4 + 1 =< A3, A3 =< A2 + 4 + 3,                            (4b)

A2 + 4 =< A4, A4 =< A2 + 4 + 3,                                (4c)

A3 + 3 + 3 =< A5, A4 + 5 + 3 =< A5,                            (4d)

B1 + 5 =< B2, B2 + 4 =< B3, B3 + 6 =< B4, B4 + 4 =< B5,        (5)

cumulative([A1,A5,B1,B3],[Da1,4,5,6],[1,1,1,1],1),             (6)
cumulative([A3,B2,B4],[3,4,4],[1,1,1],1),
cumulative([A2,A4,B5],[4,5,3],[1,1,1],1),

cumulative(StartTasks,[Da1,4,3,5,4,5,4,6,4,3],
                      [Pa1,2,1,3,1,2,1,2,3,2],4) .             (7)
```

Zuerst wird den Variablen eine endliche Domäne aus dem Bereich der natürlichen Zahlen zugeordnet. Die Bedingung 3 ist auch in der Domänenwahl der Startzeitvariablen enthalten. Für die Modellierung des Problems wurden nur vordefinierte Prädikate verwendet. Mit kleinen syntaktischen Änderungen kann diese Modellierung sofort in CHIP eingegeben werden.

Das symbolische Constraint `element(N,List,Wert)` gilt, wenn `Wert` identisch mit dem `N`'ten Element der Liste `List` natürlicher Zahlen ist. Das Constraint `notin(X,From,To)` gilt, wenn in der Domäne von `X` nicht die Werte `From` bis `To` vorkommen. Das Constraint `cumulative/4` wurde bereits oben erläutert.

Der Constraint-Löser streicht aus den Domänen der Variablen Elemente, die nicht zu einer Lösung gehören können. Zu den Domänen der Variablen können aber trotzdem noch Elemente gehören, die in einer Lösung nicht vorkommen können. Ein Constraint-Löser mit dieser Arbeitsweise wird auch als *nicht vollständig* bezeichnet. Nach dem Absetzen der Constraints

des gegebenen Beispiels, konnte der Constraint-Löser die Domänen der Startzeitvariablen wie folgt beschränken:

```
A1 in {0,1,5..8}, A2 in {3..11}, A3 in {9..12,15..17},
A4 in {7..9,15}, A5 in {15..23},
B1 in {0..5}, B2 in {5..10}, B3 in {9,12..14},
B4 in {15..20}, B5 in {19..24}
```

Für die Erzeugung einer Lösung ist eine Suche erforderlich. Dabei können schrittweise den Constraintvariablen Werte aus ihren Domänen zugeordnet werden. Mit der Zuordnung eines Wertes wird der Constraint-Löser aktiv, und die Konsequenzen werden berechnet. Falls die Domäne einer Variable bei diesen Berechnungen leer wird, ergibt sich ein Widerspruch, Backtracking wird durchgeführt und ein anderer Wert der entsprechenden Constraintvariable wird probiert. Anstatt einer Wertzuweisung kann auch eine geeignete schrittweise Domäneneinschränkung bei der Suche verwendet werden.

Wenn in unserem Beispiel bei der Suche `A1 < 5` oder `A1 > 5` hinzugefügt wird, erkennt der Constraint-Löser, daß dann nicht alle Constraints erfüllt werden können. Folglich kann für diese beiden Einschränkungen von `A1` keine Lösung existieren. Wenn nun `A1 = 5` gewählt wird, berechnet der Constraint-Löser die Konsequenzen aus dieser Entscheidung und reduziert den Suchraum drastisch, so daß sich dann als Zwischenlösung

```
A1 = 5, A2 in {9..11}, A3 in {15..17}, A4 = 15, A5 = 23,
B1 = 0, B2 = 5, B3 = 9, B4 = 20, B5 = 24,
Pa1 = 2, Da1 = 4
```

ergibt. Unter Beachtung der unter (4b) genannten Constraints erhält man dann leicht die 7 verschiedenen Lösungen des kleinen Problembeispiels.

Die Lösungssuche bei größeren Problemen ist nicht so leicht realisierbar, wie in unserem kleinen Beispiel. Bei größeren Problemen sind für eine erfolgreiche Lösungssuche Heuristiken unter Einbeziehung von Erfahrungswissen erforderlich. Durch die Methoden der Suchraumeinschränkung, die in der constraintbasierten Programmierung verwendet werden, wird das schnelle Finden von Lösungen, die nahe dem Optimum liegen, bereits bei relativ einfachen Suchheuristiken ermöglicht.

6 Vergleich von Methoden der Lösungssuche

Verschiedene Suchmethoden werden in diesem Abschnitt an einem einfachen Planungsproblem diskutiert. Gegeben seien 5 Aufträge, die jeweils aus 5 Arbeitsaufgaben bestehen. Die Arbeitsaufgaben eines Auftrages sind nacheinander auszuführen. Alle Arbeitsaufgaben haben die gleiche Bearbeitungsdauer, so daß angenommen werden kann, daß die Bearbeitungdauer jeweils eine Zeiteinheit beträgt. Zur Erfüllung einer Arbeitsaufgaben wird genau eine der 5 Maschinen benötigt. Diese Maschinenzuordnung wird durch die folgende Tabelle gegeben:

Arbeits-	Aufträge				
aufgabe	1	2	3	4	5
1	Ma 1	Ma 2	Ma 5	Ma 3	Ma 1
2	Ma 2	Ma 4	Ma 1	Ma 5	Ma 4
3	Ma 3	Ma 1	Ma 4	Ma 2	Ma 3
4	Ma 4	Ma 5	Ma 3	Ma 1	Ma 5
5	Ma 5	Ma 3	Ma 2	Ma 4	Ma 2

Aus der Tabelle ist beispielsweise zu entnehmen, daß für die Bearbeitung der 4. Aufgabe des 2. Auftrages die 5. Maschine benötigt wird. Das Ziel besteht nun darin, den Arbeitsaufgaben gegebene Zeiteinheiten so zuzuordnen, daß die Reihenfolge der Bearbeitung eingehalten wird und die Zeiteinheiten der Arbeitsaufgaben, die auf der gleichen Maschine zu bearbeiten sind, jeweils paarweise verschieden sind. Für dieses Problem existiert eine Lösung, bei der alle Aufträge innerhalb von 6 Zeiteinheiten bearbeitet werden können. In der folgenden Tabelle wird diese Lösung dargestellt:

Arbeits-	Aufträge				
aufgabe	1	2	3	4	5
1	2	1	1	1	1
2	3	3	3	2	2
3	4	4	4	4	3
4	5	5	5	5	4
5	6	6	6	6	5

Im folgenden diskutieren wir verschiedene Suchmethoden zum Erzeugen einer solchen Lösung. Um verschiedene Suchmethoden sinnvoll vergleichen zu können, wurde das Beispiel gerade so gewählt, daß die 1. Arbeitsaufgabe des 1. Auftrages in der Zeiteinheit 2 zu erfüllen ist. Außerdem wird vorausgesetzt, daß die Suche damit beginnt, daß zuerst dieser Arbeitsaufgabe die Zeiteinheit 1 zugeordnet wird. Für den Vergleich der verschiedenen Suchmethoden wird dann entscheidend sein, wie schnell die "falsche" erste Zuordnung erkannt wird.

Ein sehr naives Vorgehen der Lösungssuche besteht darin, den 25 Arbeitsaufgaben zuerst Zeiteinheiten aus $\{1,2,3,4,5,6\}$ zuzuordnen und dann erst zu überprüfen, ob die Bedingungen

erfüllt sind. Wenn dies nicht der Fall ist, wird eine andere Zuordnung erzeugt und dann wird wieder die Überprüfung der Bedingungen durchgeführt. Dies wird solange fortgesetzt bis die Lösung gefunden wird. Bereits bei diesem kleinen Problembeispiel ist aber der Suchraum so groß, daß eine Lösung nicht innerhalb von 100 Jahren gefunden werden kann, falls in einer Sekunde 10 000 Zuordnungen erzeugt und überprüft werden können. Auch wenn dabei den Arbeitsaufgaben, die auf der gleichen Maschine zu bearbeiten sind, paarweise verschiedene Zeiten zugeordnet werden, ist eine Lösung erst in etwa 100 Jahren zu erwarten. Folglich ist bereits bei diesem kleinen Problem eine intelligentere Methode der Lösungssuche erforderlich.
Für dieses Problem können leicht Zuordnungen der Zeiteinheiten aus $\{1,2,3,4,5,6\}$ so erzeugt werden, daß die Bedingungen über die Reihenfolge der Arbeitsaufgaben erfüllt sind. Wir setzen im folgenden voraus, daß nur solche Zuordnungen erzeugt werden. Somit kann auch eine Überprüfung der Bedingungen der Reihenfolge entfallen und es ist nur noch zu überprüfen, ob für jede Maschine die Zeiteinheiten, die den Arbeitsaufgaben zugeordnet sind, die diese Maschine erfordern, paarweise verschieden sind.
Um die verschiedenen Suchmethoden vergleichen zu können, wurden sie in der gleichen Programmiersprache (CHIP) implementiert. Die Problembeschreibung kann in CHIP wie folgt repräsentiert werden:

```
problem(Jobs,[Ma1,Ma2,Ma3,Ma4,Ma5],[1,2,3,4,5,6]) :-
   Jobs = [Job1,Job2,Job3,Job4,Job5],
   Job1 = [T11,T12,T13,T14,T15],    Ma1 = [T11,T23,T32,T44,T51],
   Job2 = [T21,T22,T23,T24,T25],    Ma2 = [T12,T21,T35,T43,T55],
   Job3 = [T31,T32,T33,T34,T35],    Ma3 = [T13,T25,T34,T41,T53],
   Job4 = [T41,T42,T43,T44,T45],    Ma4 = [T14,T22,T33,T45,T52],
   Job5 = [T51,T52,T53,T54,T55],    Ma5 = [T15,T24,T31,T42,T54].
```

Im folgenden werden nun vier verschiedene Suchmethoden und die Definitionen der entsprechenden Startprädikate angeben.

1. *Für alle Arbeitsaufgaben wird zuerst eine Zuordnung der Zeiteinheiten erzeugt. Anschließend erfolgt die Überprüfung der Bedingungen. Wenn nicht alle Bedingungen erfüllt sind, wird Backtracking durchgeführt und die Suche wird mit einer anderen Zuordnung fortgesetzt.*

```
solve_1 :-
   problem(Jobs,MaTasks,Numbers),
   gen_values_to_jobs(Jobs,Numbers),
   all_different(MaTasks),
   write(result(Jobs)).
```

2. *Zuerst wird eine Zuordnung der Zeiteinheiten für die ersten beiden Aufträge erzeugt. Dann wird überprüft, ob die entsprechenden Bedingungen für diese eingeschränkte Zuordnung erfüllt sind. Wenn dies nicht der Fall ist, wird eine andere Zuordnung für die beiden Aufträge erzeugt und anschließend wieder die Bedingungen überprüft. Dies wird fortgesetzt bis für diese beiden Aufträge eine widerspruchfreie Zuordnung gefunden wurde. Dann wird in analoger Weise versucht, eine Zuordnung für den dritten Auftrag so zu erzeugen, daß die relevanten Bedingungen für die drei ersten Aufträge erfüllt sind. Wenn eine solche Zuordnung für den dritten Auftrag nicht gefunden werden kann, erfolgt Backtracking und eine andere widerspruchsfreie Zuordnung der ersten beiden Aufträge wird wie oben beschrieben erzeugt. Anschließend wird wieder versucht, eine widerspruchsfreie Erweiterung der Zuordnung (für den dritten Auftrag) zu erzeugen. Bis für alle Arbeitsaufgaben eine widerspruchsfreie Zuordnung der Zeiteinheiten gefunden wurde, wird die Suche in dieser Weise fortgesetzt.*

```
solve_2 :-
   problem(Jobs,MaTasks,Numbers),
   Jobs = [Job1,Job2,Job3,Job4,Job5],
   gen_values_to_tasks(Job1,Numbers),
   gen_values_to_tasks(Job2,Numbers),
   test_different(MaTasks,2),
   gen_values_to_tasks(Job3,Numbers),
   test_different(MaTasks,3),
   gen_values_to_tasks(Job4,Numbers),
   test_different(MaTasks,4),
   gen_values_to_tasks(Job5,Numbers),
   test_different(MaTasks,5),
   write(result(Jobs)).
```

3. *Die Bedingungen werden als Constraints betrachtet und ein Constraint-Löser wird verwendet, um die Erfüllung der Constraints zu sichern. Folglich können zuerst die Constraints abgesetzt werden. Die Zuordnung der Zeiteinheiten wird dann anschließend erzeugt. Wenn der Constraint-Löser während der Zuordnung einen Widerspruch erkennt, erfolgt sofort Backtracking und die Zuordnung wird modifiziert. Bemerkenswert ist, daß sich das Programm für diese Methode von dem Programm* `solve_1` *der 1. Methode nur im Vertauschen zweier Prozeduraufrufe unterscheidet.*

```
solve_3 :-
   problem(Jobs,MaTasks,Numbers),
   all_different(MaTasks),
   gen_values_to_jobs(Jobs,Numbers),
   write(result(Jobs)).
```

4. *Ein Constraint-Löser für endliche Wertebereiche wird verwendet. Den Arbeitsaufgaben werden zuerst als Wertebereiche (Domänen) jeweils die Zeiteinheiten {1,2,3,4,5,6} zugeordnet. Dann werden die Bedingungen, die wieder als Constraints betrachtet werden, abgesetzt. Da bei dieser Methode die Wertezuordnung anders erfolgt, sind hier, im Unterschied zu den drei vorhergehenden Verfahren, auch die Constraints über die Reihenfolge der Arbeitsaufgaben abzusetzen. Der Constraint-Löser streicht nun Elemente aus den Domänen, die nicht in einer Lösung enthalten sein können. Die Lösungssuche besteht nun darin, die restlichen Werte aus den Domänen schrittweise "auszuprobieren". Der Constraint-Löser berechnet bei jeder solchen Wertezuweisung die Konsequenzen und streicht eventuell weitere Elemente aus den Domänen. Wenn der Constraint-Löser einen Widerspruch (eine leere Domäne) erkennt, erfolgt sofort Backtracking und ein andere Zuordnung wird erzeugt.*

```
solve_4 :-
    problem(Jobs,MaTasks,Numbers),
    gen_domains(Jobs,Numbers),
    task_order(Jobs),
    all_different_fd(MaTasks),
    select_value(Jobs),
    write(result(Jobs)).
```

In der folgenden Tabelle sind die Laufzeiten dargestellt, die die verschiedenen Methoden für die Lösungssuche auf dem gleichen Computersystem benötigten.

Methode:	1	2	3	4
cpu-Zeit:	11 900 ms	3 900 ms	1 050 ms	10 ms

Die Suchraumgröße ist bei diesen vier Methoden vergleichbar. In allen Fällen können den Arbeitsaufgaben potentiell 2 verschiedene Zeiteinheiten zugeordnet werden. Auch bei der 4. Methode enthalten die Domänen nach dem Absetzen der Constraints jeweils genau zwei Elemente. Das schnelle Erkennen von Wertezuweisungen, die zu keiner Lösung führen können, ist für die Lösungssuche entscheidend. Die Vorteile der constraintbasierten Methoden zeigen sich deutlich. Die Ergebnisse dieses Vergleiches können auch verallgemeinert werden. Bei anderen Suchproblemen sind ähnliche Ergebnisse zu erwarten.
Suchprobleme können natürlich auch erfolgreich in konventionellen Programmiersprachen implementiert werden. So könnte bei dem diskutierten Problembeispiel die 2. Methode so verfeinert werden, daß nach jeder Wertezuweisung eine Überprüfung der Bedingungen erfolgt. Dann sind auch Ergebnisse zu erwarten, die mit der Verwendung eines Constraint-Lösers erreicht

werden. Aber der Programmieraufwand einer solchen Implementation in einer konventionellen Programmiersprache ist wesentlich größer. Bereits in der diskutierten Implementation war die Programmierung der 2. Methode am aufwendigsten. Bei etwas komplexeren Bedingungen (bereits wie bei dem im 4. Abschnitt diskutierten Beispiel) ist fraglich, ob nach jeder Wertezuweisung überhaupt eine hinreichende Überprüfung der Bedingungen realisierbar ist. Ein weiterer Nachteil besteht darin, daß bei Veränderungen der Bedingungen auch die Prozeduren für die Überprüfungen entsprechend verändert werden müssen. Bei der constraintbasierten Programmierung bleibt der Constraint-Löser, der die Erfüllung der Bedingungen sichert, dagegen unverändert.

7 Literaturhinweise

Die constraintlogische Programmierung wurde erstmalig in [JL87] definiert. Einführungen in diese Programmierung sind beispielsweise [FHK+92] und [HSD92]. Die ersten Lehrbücher der constraintlogischen Programmierung sind [MS98] und [FA97]. Eine Übersicht zu den Grundlagen der constraintlogischen Programmierung wird in [JM94] gegeben. Grundlagen der Constraintverarbeitung werden in [Tsa93] dargestellt.
Die constraintlogischen Programmiersprachen PROLOG III, CLP($\mathcal{R}$) und CHIP werden in [Col87], [JM87] bzw. [DvHS+88] kurz vorgestellt. Spezielle Inferenzregeln für Constraints über endlichen Domänen, die u.a. auch in CHIP integriert sind, werden in [Hen89] ausführlich beschrieben. Die Einbeziehung von globalen Constraints in CHIP wird in [AB93] und [BC94] diskutiert.
Aktuelle Forschungsergebnisse aus dem Gebiet der constraintbasierten Programmierung sind beispielsweise in den folgenden Sammelwerken enthalten: [MTP94], [Jou94], [BC93], [Bor94], [MR95], [Pod95], [Fre96]. Viele verschiedene praktische Anwendungen der constraintbasierten Programmierung wurden in den letzten Jahren auf den internationelen Konferenzen *Practical Application of Constraint Technology* und *Practical Application of Prolog* vorgestellt. Praktische Anwendungen der constraintbasierten Programmierung werden auch in [Wal96] diskutiert. Eine der ersten Arbeiten, die das Lösen von Problemen mittels der constraintlogischen Programmierung diskutierte, war [DSvH90] .
In unserer Forschungsgruppe sind u.a. Arbeiten zu den folgenden Schwerpunkten der Anwendung der constraintlogischen Programmierung entstanden: Behandlung von Planungsproblemen mit constraint-logischen Methoden ([GSS96], [Ges95], [Nit94], [For96], [GM99]), Verfahren zur Suchraumeinschränkung beim Lösen von Problemen ([Gol95], [GJ96], [Gol96], [Joh96], [Nar95], [NG96]), praktische Anwendungen von Constrainthierarchien ([Wol96b, Wol96a]).

Literatur

[AB93] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *J. Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[BC93] F. Benhamou and A. Colmerauer, editors. *Constraint Logic Programming (Selected Research).* MIT Press, Cambridge, London, 1993.

[BC94] E. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *J. Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[Bor94] A. Borning, editor. *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, 1994.

[Col87] A. Colmerauer. Opening the PROLOG III universe. *BYTE*, pages 177 – 182, August 1987.

[DSvH90] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving large combinatorial problems in logic programming. *J. Logic Programming*, 8:75–93, 1990.

[DvHS$^+$88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Int. Conf. Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, 1988.

[FA97] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung.* Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[FHK$^+$92] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming – an informal introduction. In G. Comyn, N. E. Fuchs, and M. J. Ratcliffe, editors, *Logic Programming in Action*, volume 636 of *Lecture Notes in Artificial Intelligence*, pages 3–35, Berlin, Heidelberg, 1992. Springer-Verlag.

[FK94] E.C. Freuder and K.Mackworth, editors. *Constraint-Based Reasoning.* MIT Press, 1994.

[For96] A. Fordan. Optimierung eines Constraint-intensiven Problems durch frühe Projektion. In *Workshop Deklarative Constraint-Programmierung während der KI-Konferenz'96*, pages 59–74. GMD-Studien 297, September 1996.

[Fre96] E.C. Freuder, editor. *Principles and Practice of Constraint Programming - CP'96*, volume 1118 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, 1996.

[Ges95] U. Geske. Optimization with heuristic-controlled constraint-logic methods. *Systems Analysis, Modelling, Simulation. A journal of mathematical modelling and simulation in systems analysis*, 18–19, Juni 1995. Gordon and Breach Publisher.

[GJ96] H.-J. Goltz and U. John. Methods for solving practical problems of job-shop scheduling modelled in CLP(FD). In *PACT 96 – Practical Application of Constraint Technology*, pages 73–92, 1996.

[GM99] H.-J. Goltz and D. Matzke. University timetabling using constraint logic programming. In G. Gupta, editor, *Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 320–334, Berlin, Heidelberg, New York, 1999. Springer-Verlag.

[Gol95] H.-J. Goltz. Reducing domains for search in CLP(FD) and its application to job-shop scheduling. In *[MR95]*, pages 549–562, 1995.

[Gol96] H.-J. Goltz. Anwendung heuristischer Verfahren zur Suchraumeinschränkung beim Lösen von Problemen der Ablaufplanung. In J. Sauer, A. Günter, and J. Hertzberg, editors, *Planen und Konfigurieren 96*, volume 3 of *Proceedings in Artificial Intelligence*, pages 111–117, Sankt Augustin, 1996. Infix.

[GSS96] U. Geske and A. Schmücker-Schend. Behandlung von Planungsproblemen mit constraint-logischen Methoden. In J. Sauer, A. Günter, and J. Hertzberg, editors, *Planen und Konfigurieren 96*, volume 3 of *Proceedings in Artificial Intelligence*, pages 31–45, Sankt Augustin, 1996. Infix.

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge (Mass.), London, 1989.

[HSD92] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992. also in [FK94].

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th Principles of Programming Languages*, pages 111–119, Munich, 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez, editor, *Proc. 4th Int. Conf. Logic Programming*, pages 196–218. MIT Press, Cambridge (Mass.), London, 1987.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Logic Programming*, 19/20:503–581, 1994.

[Joh96] U. John. Zerlegung großer Planungsprobleme. In J. Sauer, A. Günter, and J. Hertzberg, editors, *Planen und Konfigurieren 96*, volume 3 of *Proceedings in Artificial Intelligence*, pages 130–136, Sankt Augustin, 1996. Infix.

[Jou94] J.-P. Jouannaud, editor. *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1994.

[MR95] U. Montanari and F. Rossi, editors. *Principles and Practice of Constraint Programming - CP'95*, volume 976 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1995.

[MS98] K. Marriott and P. J. Stucky. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge (MA), London, 1998.

[MTP94] B. Mayoh, E. Tyugu, and J. Penjam, editors. *Constraint Programming*. NATO ASI Series. Springer-Verlag, Berlin, Heidelberg, 1994.

[Nar95] A. Nareyek. Modellierung der Disjunktion in der constraintlogischen Programmierung. Arbeitspapiere der GMD 932, GMD – Forschungszentrum Informationstechnik GmbH, 1995.

[NG96] A. Nareyek and U. Geske. Behandlung expliziter Alternativen in CLP(FD). In *Workshop Deklarative Constraint-Programmierung während der KI-Konferenz'96*, pages 75–85. GMD-Studien 297, September 1996.

[Nit94] M. Nitsche. Analysis and refinement of constraint answer sets in a planning system. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *GULP-PRODE '94, Joint Conference on Declarative Programming*, pages 18–30, 1994.

[Pod95] A. Podelski, editor. *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1995.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[Wal96] M. Wallace. Practical applications of contraint programming. *Constraints, An International Journal*, 1:139–168, 1996.

[Wol96a] A. Wolf. Flexiblere Planung mit Constrainthierarchien. In J. Sauer, A. Günter, and J. Hertzberg, editors, *Planen und Konfigurieren 96*, volume 3 of *Proceedings in Artificial Intelligence*, pages 104–110, Sankt Augustin, 1996. Infix.

[Wol96b] A. Wolf. Transforming ordered constraint hierarchies into ordinary constraint systems. In M. Jampel, E. Freuder, and M. Maher, editors, *Over-Constrained System*, volume 1106 of *Lecture Notes in Computer Science*, pages 171–187, Berlin, Heidelberg, 1996. Springer-Verlag.

Anhang

Als Ergänzung zu den Programmen, die im 6. Abschnitt für die dort diskutierten Suchmethoden gegebenen wurden, werden im folgenden die Definitionen der in diesen Startprädikaten verwendeteten Prozeduren, die nicht vordefiniert sind, gegeben. Damit sind die Programme der diskutierten Suchmethoden vollständig angegeben und können ohne Änderungen in der constraintlogischen Programmiersprache CHIP verwendet werden.

```
gen_values_to_jobs([],_).
gen_values_to_jobs([Tasks|Jobs],Ns) :-
   gen_values_to_tasks(Tasks,Ns),
   gen_values_to_jobs(Jobs,Ns).

gen_values_to_tasks([],_).
gen_values_to_tasks([Task|Tasks],Ns) :-
    select(Task,Ns,NsRest),
    gen_values_to_tasks(Tasks,NsRest).

select(X,[X|Rest],Rest).
select(X,[_|List],Rest) :-
    select(X,List,Rest).
```

```
all_different([]).
all_different([Tasks|MaTasks]) :-
  different(Tasks),
  all_different(MaTasks).

different([]).
different([_]).
different([X|List]) :-
  different1(List,X),
  different(List).

different1([],_).
different1([Y|List],X) :-
  X =\= Y,
  different1(List,X).
```

```
test_different([],_).
test_different([Tasks|MaTasks],N) :-
  del(N,1,Tasks,List,Task),
  different1(List,Task),
  test_different(MaTasks,N).

gen_domains([],_).
gen_domains([Job|Jobs],Ns) :-
  Job::Ns,
  gen_domains(Jobs,Ns).

all_different_fd([]).
all_different_fd([Tasks|MaTasks]) :-
  alldifferent(Tasks),
  all_different_fd(MaTasks).

select_value([]).
select_value([VarList|Rest]) :-
   select_value1(VarList),
   select_value(Rest).
```

```
del(N,N,[Task|_],[],Task) :- !.
del(N,M,[T1|Ts],[T1|List],Task) :-
    M < N,
    M1 is M + 1,
    del(N,M1,Ts,List,Task).

task_order([]).
task_order([Job|Jobs]) :-
   task_order1(Job),
   task_order(Jobs).

task_order1([_]).
task_order1([T1,T2|Tasks]) :-
   T1 #< T2,
   task_order1([T2|Tasks]).

select_value1([]).
select_value1([Var|VarList]) :-
   indomain(Var),
   select_value1(VarList).
```