# Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz

**Dissertation**

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät
der Universität des Saarlandes

von
Diplom-Informatiker
**Jörg Würtz**

Saarbrücken
Januar 1998

| | |
|---|---|
| Tag des Kolloquiums: | 14. Mai 1998 |
| Dekan: | Professor Dr. Alexander Koch |
| Gutachter: | Professor Dr. Gert Smolka |
| | Priv.-Doz. Alexander Bockmayr |

# Zusammenfassung

In dieser Dissertation beschäftigen wir uns mit der Lösung kombinatorischer Probleme durch Constraintprogrammierung. Wir zeigen, daß verschiedene kombinatorische Probleme in der nebenläufigen Constraintsprache Oz effizient gelöst werden können. Wir führen ein formales Modell von constraintbasiertem Lösen kombinatorischer Probleme ein, das unabhängig von einer konkreten Programmiersprache ist, und wir zeigen, wie einige der derzeit besten Schedulingtechniken (Techniken für Ablaufplanung) aus dem Operations Research für Constraintpropagierung und Distribuierung in dieses Modell integriert werden können. Wir zeigen, wie dieses Modell in die nebenläufige Constraintsprache Oz eingebettet werden kann und belegen mit einer Reihe von Fallstudien für große und schwierige Probleme aus dem Gebiet des Scheduling die Leistungsfähigkeit des entwickelten Systems.

# Abstract

In this doctoral dissertation we deal with combinatorial problem solving by constraint programming. We show that a number of combinatorial problems can be efficiently solved in the concurrent constraint language Oz. We introduce a formal model for constraint-based combinatorial problem solving which is independent from a particular programming language. We show how some of the currently best scheduling techniques from Operations Research can be integrated in the model of constraint propagation and distribution. We embed this model in the concurrent constraint language Oz and prove the efficiency of the developed system by a series of case studies for large and hard problems from the area of scheduling.

# Ausführliche Zusammenfassung

In dieser Dissertation beschäftigen wir uns mit der Lösung kombinatorischer Probleme durch Constraintprogrammierung. Beispiele für kombinatorische Probleme sind das Ermitteln eines Ablaufplans für eine Produktion (Scheduling), das Erstellen eines Schichten- oder Stundenplans oder die Konfiguration technischer Systeme. Ein kombinatorisches Problem kann durch eine Menge von Constraints und eine Menge von Variablen über endlichen Bereichen ganzer Zahlen beschrieben werden. Die Menge aller möglichen Belegungen der Variablen definiert den Suchraum für ein Problem. Eine Lösung eines kombinatorischen Problems ist eine Belegung jeder Variablen mit einem Wert aus dem ihr zugehörigen Bereich, so daß alle Constraints erfüllt sind.

Mit Constraintprogrammierung können viele kombinatorische Probleme sowohl elegant modelliert als auch effizient gelöst werden. Dabei werden durch Propagierung Werte aus dem Bereich einer Variablen entfernt, die in keiner Lösung als Belegung dieser Variablen vorkommen können. Durch sogenannte Distribuierung wird die Lösungssuche gesteuert.

In dieser Arbeit zeigen wir, daß verschiedene kombinatorische Probleme in der nebenläufigen Constraintsprache Oz effizient gelöst werden können. Oz ist eine nebenläufige Constraintsprache, die auch funktionales und objektorientiertes Programmieren unterstützt. Zusätzlich bietet Oz die Möglichkeit, neue Suchstrategien zu programmieren.

Wir führen ein formales Modell von constraintbasiertem Lösen kombinatorischer Probleme ein, das unabhängig von einer konkreten Programmiersprache ist, und wir zeigen, wie einige der derzeit besten Schedulingtechniken aus dem Operations Research für Propagierung und Distribuierungsstrategien in dieses Modell integriert werden können. Wir zeigen, wie dieses Modell in die nebenläufige Constraintsprache Oz eingebettet werden kann und belegen mit einer Reihe von Fallstudien für große und schwierige Probleme aus dem Gebiet des Scheduling die Leistungsstärke des entwickelten Systems.

Beiträge wurden geleistet in dem Bereich Modellierung, in dem Anwendungsgebiet Scheduling sowie im Design und in der Implementierung von Programmiersprachen.

## Modellierung von constraintbasiertem Lösen kombinatorischer Probleme

Wir stellen ein formales Modell von constraintbasiertem Lösen kombinatorischer Probleme vor, das unabhängig von einer konkreten Programmiersprache ist. Dieses Modell beruht auf Propagierung und Distribuierung. Bei der Entwicklung des Modells berücksichtigen wir, daß es als Grundlage für eine effiziente Implementierung dienen soll.

In dieses formale Modell integrieren wir einige der derzeit besten Techniken aus dem Operations Research zur Lösung von disjunktiven und kumulativen Schedulingproblemen. Dies sind zum einen Techniken, um die Propagierung für diese spezielle Art von Problemen zu verstärken (Edge-Finding). Zum anderen sind dies Distribuierungsstrategien, die die spezielle Struktur von Schedulingproblemen ausnutzen, um die Suche zu steuern.

**Design und Implementierung von Programmiersprachen**

Wir integrieren das Modell von Propagierung in das Berechnungsmodell von Oz. Dabei legen wir besonderen Wert auf Kompatibilität mit den übrigen Sprachkonstrukten von Oz sowie auf gute Benutzbarkeit und Expressivität. Wir zeigen insbesondere auch, wie die in unser Modell integrierten Schedulingtechniken in Oz übernommen werden können.

Wir geben eine ausführliche Darstellung eines Implementierungsmodells für Propagierung in einer nebenläufigen Constraintsprache an. Wir beschreiben sowohl die prinzipielle Integration des Propagierungsmodells in Oz als auch notwendige Optimierungen, um eine effiziente Ausführung zu erreichen. Ein weiterer Schwerpunkt liegt auf Erweiterungen, mit denen die Implementierung von Constraints für Schedulinganwendungen unterstützt wird.

## Constraintbasiertes Scheduling

Wir übertragen eine neue Schedulingtechnik für disjunktive Schedulingprobleme aus dem Operations Research in die Constraintprogrammierung. Ausgehend von der Originalarbeit verstärken wir die Propagierung und verallgemeinern die Technik auf kumulative Schedulingprobleme.

Wir lösen erstmals mit Constraintprogrammierung Probleme aus einem neuen Anwendungsgebiet der Autoelektronik sowie der Luft- und Raumfahrtindustrie (taktgesteuerte Echtzeitsysteme). Diese Probleme zeichnen sich durch ungewöhnliche Eigenschaften sowie durch eine enorme Problemgröße aus. Bei der Lösung entsprechender Probleme kommen die in Oz integrierten Schedulingtechniken zum Einsatz.

Zur Lösung von disjunktiven und kumulativen Schedulingproblemen realisieren wir eine interaktive, grafische und erweiterbare Werkbank. Diese Werkbank greift auf in Oz vordefinierte Funktionalität zurück, bietet aber auch eine Kombination aus Constraintprogrammierung und heuristischen Verfahren an. Der Benutzer kann in Oz verfügbare Schedulingtechniken zur Lösung seiner Probleme beliebig kombinieren und die Probleme interaktiv lösen.

Wir nehmen eine umfassende Evaluierung von Schedulingtechniken für Job-Shop-Probleme vor. Dabei werden Optimalitätsbeweise von Lösungen, das Finden möglichst guter oder optimaler Lösungen aber auch Verfahren zur Bestimmung von unteren Schranken verwendet. Außerdem analysieren wir die für Scheduling wichtigen Entwurfsentscheidungen bei der Integration in eine Constraintsprache.

# Extended Abstract

In this doctoral dissertation we deal with combinatorial problem solving by constraint programming. Examples for combinatorial problems are scheduling for a production line, computation of a timetable or configuration of technical systems. A combinatorial problem can be represented by a set of constraints and a set of variables ranging over a finite domain of integer numbers. The set of all possible assignments for the variables defines the search space of a problem. A solution of a combinatorial problem is an assignment for each variable with a value taken from the corresponding domain such that all constraints are satisfied.

By constraint programming many combinatorial problems can be modeled and efficiently solved. By propagation values are deleted from the domain of a variable which cannot occur in a solution as an assignment of that variable. By so-called distribution the search for a solution can be guided.

In this thesis we show that a number of combinatorial problems can be efficiently solved by the concurrent constraint language Oz. Oz is a concurrent constraint language supporting functional and object-oriented programming. Additionally Oz provides for the prossibility to program new search strategies.

We introduce a formal model for constraint-based solving of combinatorial problems which is independent from a particular programming language. We show how some of the currently most successful scheduling techniques from Operations Research can be integrated in the model. We show how the model can be embedded in the concurrent constraint language Oz and we prove by a series of case studies for large and hard scheduling problems the efficiency of the developed system.

We made contributions in modelling, in the application area scheduling and in the design and implementation of programming languages.

## Modeling of constraint-based combinatorial problem solving

We introduce a formal model for constraint-based combinatorial problem solving which is independent of a particular programming language. This model is based on propagation and distribution. For the development of the model we take care of the fact that it should be the starting-point for an efficient implementation.

In this formal model we integrate some of the currently most successful techniques from Operations Research for solving disjunctive and cumulative scheduling problems. On the one hand, these are techniques (edge-finding) to strengthen the propagation for this particular class of problems. On the other hand, these are distribution strategies which employ the special structure of scheduling problems to guide the search.

### Design and implementation of programming languages

We integrate the model for propagation in the computation model of Oz. Here we emphasize compatibility with the other language constructs of Oz and a high usability and expressivity. In particular we show how the scheduling techniques integrated in the model can be carried over to

Oz.

We give a comprehensive overview of an implementation model for propagation in a concurrent constraint language. We describe the essential integration of the propagation model in Oz as well as the neccessary optimizations to achieve an efficient execution. We further emphasize some extensions to support the implementation of constraints for scheduling applications.

## Constraint-based scheduling

We carry a new scheduling technique for disjunctive scheduling problems from Operations Research over to constraint programming. Starting from the original work we strengthen the propagation and generalize the technique for cumulative scheduling problems.

We solve for the first time problems from a new application area in automotive electronics and aerospace industry by constraint programming (time-triggered real-time systems). These problems show uncommon characteristics and an enormous problem size. For solving such problems scheduling techniques are employed which are integrated in Oz.

To solve different scheduling problems, we implement an interactive, graphical and extendible workbench. This workbench uses predefined functionality in Oz but provides also for a combination of constraint programming with an heuristic approach. The user may freely combine scheduling techniques available in Oz to solve problems interactively.

We provide a comprehensive evaluation of scheduling techniques for job-shop problems. For this we use proofs of optimality, search for good or optimal solutions or techniques to find lower bounds. Moreover we analyze the design decisions which are relevant for integrating scheduling techniques into a constraint language.

*Für Patricia*

# Danksagung

Zuerst möchte ich allen Mitgliedern des Forschungsbereichs Programmiersysteme danken. Ohne ihre stete Arbeit an unserem Programmiersystem Oz, hätte diese Dissertation nicht entstehen können.

Meinem Doktorvater Gert Smolka danke ich für die vielen Ideen, die der Constraintprogrammierung in Oz immer wieder wichtige Impulse gegeben haben.

Martin Henz, Martin Müller, Tobias Müller, Joachim Niehren, Christian Schulte und Joachim Walser haben durch Diskussionen über Constraintprogrammierung diese Arbeit immer wieder stimuliert.

Tobias Müller hat das FD-System in die abstrakte Maschine von Oz integriert und viele Propagierer der FD-Bibliothek entwickelt und implementiert. Ohne seinen Beitrag zum Design und zur Implementierung des FD-Systems von Oz wäre diese Dissertation nicht möglich gewesen.

Martin Henz danke ich für seine Mitarbeit an unserem ersten großen Programm zum Lösen eines kombinatorischen Problems.

Klaus Schild danke ich für seine Zusammenarbeit bei unserem Projekt über taktgesteuerte Echtzeitsysteme.

Björn Carlson, Mats Carlsson, Yves Caseau, Daniel Diaz, François Laburthe, Paul Martin, Wim Nuijten, Pascal Van Hentenryck und Jianyang Zhou danke ich für viele Diskussionen über Constraintprogrammierung und Scheduling.

Martin Müller, Tobias Müller, Christian Schulte, Ralf Scheidhauer, Gert Smolka und Joachim Walser danke ich für Kommentare zu früheren Versionen dieser Arbeit.

Zum Schluß möchte ich ganz besonders meiner Frau Patricia für ihre Geduld in den letzten Monaten danken.

Jörg Würtz
Saarbrücken
Januar 1998

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation und Forschungsgebiet

In dieser Arbeit untersuchen wir constraintbasiertes Lösen von kombinatorischen Problemen.

### 1.1.1 Kombinatorische Probleme

In dieser Arbeit beschäftigen wir uns mit dem Lösen kombinatorischer Probleme. Beispiele solcher Probleme sind das Ermitteln eines Ablaufplans für eine Produktion (Scheduling), das Erstellen eines Schichten- oder Stundenplans oder die Konfiguration technischer Systeme. Das Lösen solcher Probleme ist eine Voraussetzung dafür, die zunehmende Komplexität in der realen Welt zu beherrschen.

Ein kombinatorisches Problem kann durch eine Menge von Variablen und eine Menge von Bedingungen beschrieben werden, die zwischen den Variablen gelten müssen. Für solche Bedingungen verwenden wir den Begriff *Constraints*. Die Variablen dürfen nur ganzzahlige Werte aus einem *endlichen Bereich* (engl. *finite domain*) annehmen. Eine *Lösung* eines kombinatorischen Problems ist eine Belegung jeder Variablen mit einem Wert aus dem ihr zugehörigen Bereich, so daß alle Constraints erfüllt sind. Die Menge aller möglichen Belegungen von Variablen bildet den *Suchraum*. Der Suchraum ist in der Regel bei weitem zu groß, um durch vollständiges Absuchen Lösungen zu finden [GJ79].

### 1.1.2 Nebenläufige Constraintprogrammierung

Anfang der neunziger Jahre zeigte es sich, daß mit Constraintprogrammierung viele kombinatorische Probleme sowohl elegant repräsentiert als auch effizient gelöst werden können (siehe z. B. [DSV90, AB93, DC93]). Die entwickelten Systeme verwenden *Propagierung*, um den Suchraum zu verkleinern. Dabei werden Werte aus dem Bereich einer Variablen entfernt, die in keiner Lösung des Problems als Belegung dieser Variablen vorkommen können. Durch sogenannte *Distribuierung* (oder Labeling) wird die Lösungssuche gesteuert, wobei insbesondere problemspezifisches Wissen ausgenutzt werden kann. Constraintprogrammierung zeichnet sich

aus durch eine hohe Flexibilität, durch die Möglichkeit, mächtige algorithmische Verfahren zur Propagierung in Constraints zu kapseln und zu kombinieren, und durch eine klare Trennung von Problemrepräsentation und Lösungssuche.

Ein spezieller Zweig der Constraintprogrammierung ist das relativ junge Gebiet der nebenläufigen Constraintprogrammierung [Mah87, Sar93]. Nebenläufige Constraintprogrammierung ist ein einfaches doch mächtiges Modell für nebenläufige Berechnung, das es erlaubt, Synchronisationskonstrukte wie Konditionale mit Constraintprogrammierung zu verbinden. Das Oz-Programmiermodell [Smo95] erweitert das Modell der nebenläufigen Constraintprogrammierung und ist durch die Programmiersprache Oz realisiert [Pro97]. So umfaßt das Oz-Programmiermodell auch funktionales und objekt-orientiertes Programmieren und gibt dem Benutzer durch geeignete Programmierabstraktionen die Möglichkeit, neue Suchstrategien in Oz zu implementieren [SS94, Sch98b].

In dieser Dissertation soll untersucht werden, inwieweit verschiedene kombinatorische Probleme mit der nebenläufigen Constraintsprache Oz effizient gelöst werden können. Dazu entwickeln wir ein formales Modell von constraintbasiertem Lösen kombinatorischer Probleme, das unabhängig von einer konkreten Programmiersprache ist. Wir erweitern dann das Oz-Programmiermodell und die konkrete Implementierung von Oz, indem wir Propagierung diesem Modell gemäß einbetten. Um die Effizienz des entstehenden Systems zu untersuchen, haben wir Scheduling (Ablaufplanung) als Anwendungsgebiet ausgewählt, da hier vielfältige Anwendungen gegeben und bereits Erfahrungen mit Constraintprogrammierung vorhanden sind [CL94a, Nui94, AB93]. Dazu haben wir einige der derzeit besten Schedulingtechniken aus dem Operations Research [CP89, AC91, MS96] in das Modell und dann in Oz integriert, sowie eine Reihe von Fallstudien mit großen und schwierigen Problemen [HW96a, SW98, Wür96, Law84] durchgeführt.

## 1.2 Constraintprogrammierung

Wir wollen hier *Constraints* als logische Formeln sehen, die Relationen über den ganzen Zahlen beschreiben. Ein kombinatorisches Problem kann dann durch eine Menge solcher Constraints über Variablen beschrieben werden, die nur Werte aus einem endlichen Bereich von ganzen Zahlen annehmen dürfen. Eine Lösung eines kombinatorischen Problems ist eine Belegung jeder Variablen mit einem Wert aus dem zugehörigen Bereich, so daß alle beschreibenden Constraints erfüllt sind.

Werden Constraints nur *passiv* benutzt, nämlich um herauszufinden, ob einzelne Belegungen Lösungen sind, so muß der gesamte Suchraum (die Menge aller möglichen Belegungen der Variablen) exploriert werden. Die grundlegende Idee von Constraintprogrammierung ist es nun, die möglichen Werte von Variablen (also ihre Bereiche) während der Lösungssuche durch Constraints *aktiv* einzuschränken.

Wir geben in diesem Abschnitt eine kurze Einordnung der Constraintprogrammierung und der Programmiersprache Oz.

### 1.2.1 Constraintnetze

Eine aktive Rolle zur Verkleinerung des Suchraumes spielen Constraints bei den sogenannten Constraintnetzen [Mon74, Wal75, Kum92, Tsa93]. Hierbei soll herausgefunden werden, ob eine Menge von Constraints erfüllbar ist (wir vermeiden den englischen Begriff *constraint satisfaction problem solving (CSP)*, da dieser von zu allgemeiner Natur ist).

Ein kombinatorisches Problem wird durch eine Menge von Variablen und Constraints beschrieben. Eine Menge von Variablen wird genau dann im Constraintnetz durch eine einzelne (Hyper-) Kante verknüpft, wenn zwischen ihnen ein entsprechender Constraint gilt. Mit jeder Variablen ist ein *Bereich* assoziiert, der die Werte angibt, die die Variable in einer Belegung noch annehmen kann. Ein *Konsistenzalgorithmus* entfernt durch *Propagierung* Werte aus den Variablenbereichen, die nicht zu einer Lösung beitragen können [Mac77, MF85]. So können z. B. für $x + 10 \leq y$ alle Werte aus dem Bereich von $y$ entfernt werden, die kleiner als der kleinste noch mögliche Wert von $x$ plus 10 sind.

Kann der Konsistenzalgorithmus den Suchraum nicht weiter verkleinern (also keine Bereiche verkleinern), so können immer noch Variablen vorhanden sein, für die kein eindeutiger Wert festliegt. Nun wird eine Variable ausgewählt und die Variable wird mit diesem Wert belegt. Das Zusammenspiel von Propagierung und Belegung geschieht so lange, bis eine Lösung gefunden ist. Findet man vorher eine Inkonsistenz, so wird auf einen früheren Suchzustand zurückgesetzt und ein alternatives Paar aus Variable und Wert wird betrachtet (engl. Backtracking). Für dieses Vorgehen von sukzessivem Belegen von Variablen verwenden wir den allgemeineren Begriff *Distribuierung*, der später auch komplexere Techniken umfassen wird (gebräuchlich sind auch die Begriffe *Labeling*, *Enumeration* oder *Branching*). In die Variablen- und Wertauswahl gehen oft spezifische Strategien ein [HE80, Tsa93].

Um die Vorteile dieser Techniken besser nutzen zu können, sollten die Techniken in eine geeignete Programmiersprache integriert werden. Dann könnten in einem einheitlichen Rahmen auch neue Constraints und Distribuierungsstrategien implementiert werden.

### 1.2.2 Logische Constraintprogrammierung

Die Entwicklung der logischen Programmiersprache Prolog [Kow74], basierend auf Unifikation und Resolution, war der erste Schritt zur modernen Constraintprogrammierung.

In [JL87] wird erkannt, daß Unifikation von Konstruktortermen durch das Lösen von Constraints modelliert werden kann. Außerdem wird ein allgemeines Schema angegeben, wie das Lösen von Constraints mit Logikprogrammierung verknüpft werden kann. Bei jedem Resolutionsschritt wird eine Menge von Constraints erweitert und es wird getestet, ob alle Constraints dieser Menge zusammen eine Lösung besitzen können; sogenannte *globale Konsistenz* wird zugesichert. Logische Constraintprogrammierung [JM94] bietet viele Vorteile gegenüber den bis dahin in konventionellen Programmiersprachen implementierten Constraintnetzen: Logische Variablen können partielle Information repräsentieren, also insbesondere den Bereich einer Variablen, rücksetzbasierte Suche (Backtracking) ist inhärent vorhanden und muß nicht programmiert werden, automatisches Rücksetzen erlaubt die Implementierung von Distribuierungsstrategien, und durch die Verwendung von symbolischer Programmierung lassen sich Probleme leicht modellieren (mit

allen Vorteilen für die Erweiterbarkeit und Wartbarkeit von Anwendungsprogrammen). Außerdem fügen sich die deklarativen Constraints gut in das Paradigma der logischen Programmierung. Praktische Umsetzungen sind Systeme wie CLP($\mathcal{R}$) [JMSY92] (reelle Zahlen) und Prolog III [Col90] (boolesche Werte, reelle Zahlen und unendliche Bäume).

Unabhängig von den Konzepten in [JL87] wurde mit CHIP [VD87, DVS+88, Van89] ein System entwickelt, mit dem kombinatorische Probleme effizient gelöst werden können. Basierend auf Prolog, wird in CHIP das Konzept der Propagierung durch Constraintnetze aufgegriffen, in eine Programmiersprache integriert und für die Praxis erweitert. Zur Laufzeit wird ein Constraintnetz aufgebaut. Dabei werden Constraints in primitive Constraints (wie $x \in \{1, \ldots, 5\}$) und nicht-primitive Constraints unterschieden. Während für primitive Constraints globale Konsistenz sichergestellt wird, wird für nicht-primitive Constraints nur *lokale Konsistenz* (oder sogar Abschwächungen) sichergestellt (es werden Werte aus Bereichen entfernt, die nicht in einer Lösung des nicht-primitiven Constraints vorkommen können). Ein nicht-primitiver Constraint macht Information explizit, indem er seiner Semantik entsprechend primitive Constraints in das Constraintnetz einfügt. Kontrollkonstrukte wie Konditionale erlauben eine eingeschränkte Implementierung von Constraints, die nicht vordefiniert sind. Für Anwendungen können komplexe Distribuierungsstrategien implementiert werden, die nicht nur einfach eine Variablenbelegung vornehmen. Effiziente Algorithmen aus der Graphentheorie und dem Operations Research kommen bei der Implementierung sogenannter symbolischer und globaler Constraints zum Einsatz [DSV88a, DSV88b, DSV90, AB93, BC94].

Die Konzepte der logischen Constraintprogrammierung, wie sie in CHIP realisiert sind, können jedoch auch in konventionellere Programmiersprachen eingebettet werden. Eine Realisierung ist das kommerzielle System ILOG SOLVER [ILOG96b, Pug94]. Constraints und Distribuierungsstrategien werden dabei über eine C++-Klassenbibliothek zur Verfügung gestellt (siehe auch Abschnitt 13.1).

Während CHIP und ILOG SOLVER in der Praxis erfolgreich sind, gibt es für Propagierung doch nur ein (von Prolog) sprachabhängiges Berechnungsmodell [Van89]. Zudem gibt es für die wenigen ad-hoc eingeführten Konstrukte zur Implementierung neuer Constraints durch den Benutzer kein klares semantisches Modell.

### 1.2.3 Nebenläufige Constraintprogrammierung

In der nebenläufigen Logikprogrammierung [Sha89] werden komplexe Kommunikations- und Synchronisationsprotokolle mit logischen Variablen realisiert. Maher erkannte in [Mah87], daß dem Synchronisationsprinzip in der nebenläufigen Logikprogrammierung Subsumtion von Constraints zugrunde liegt. Saraswat [SR90, Sar93] entwarf darauf aufbauend die nebenläufige Constraintprogrammierung. Berechnung besteht danach aus der Aktivität von Agenten, die über einen gemeinsamen Constraintspeicher kommunizieren. Die Agenten fügen entweder Constraints zum Speicher hinzu (tell), oder warten auf das Eintreffen bestimmter Constraints (ask). Konditionale und weitere Kontrollkonstrukte kommen durch die nebenläufige Constraintprogrammierung von ihrem ad-hoc Status (wie in CHIP) los. Die Ideen der nebenläufigen Constraintprogrammierung wurden in den Programmiersystemen AKL [HJ90, Jan94] und Oz [Smo95, HSW93, HSW95, Pro97] realisiert.

Neben den allgemeinen Programmiersystemen entstanden spezielle Systeme zum Lösen von kombinatorischen Problemen. In [VSD91] werden sogenannte Indexicals eingeführt, um dem Programmierer die Implementierung eigener Constraints in gewissem Umfang zu ermöglichen. Eine praktische Umsetzung erfolgte mit `clp(FD)` [DC93, CD96], jedoch noch in einem sequentiellen System. Der Entwurf `cc(FD)` [VSD93, VSD95, Van94] ließ zwar die Idee der Indexicals fallen, bot dem Programmierer in einem nebenläufigen Szenario aber Kontrollkonstrukte wie konstruktive Disjunktion, Cardinality-Operator [VD91b] oder Konditionale zur Definition neuer Constraints an. Eine nebenläufige Implementierung von Indexicals zusammen mit Kontrollkonstrukten aus `cc(FD)` wurde mit AKL(FD) [CCJ95, Car95] realisiert. Zwar unterstützen diese Systeme die Implementierung neuer Constraints durch den Benutzer. Zur Lösung großer praktischer Probleme mangelt es ihnen jedoch an mächtigen globalen Constraints wie sie zum Beispiel CHIP zur Verfügung stellt.

### 1.2.4 Oz

Das Oz-Programmiermodell (OPM) wurde von Smolka [Smo95] als eine Erweiterung des nebenläufigen Constraintmodells vorgeschlagen. Das OPM dient als Grundlage der Programmiersprache und des Programmiersystems Oz, die am DFKI in Saarbrücken entwickelt wurden [HSW93, Pro97].

Als nebenläufige Constraintsprache bietet Oz eine große Auswahl von Kontrollkonstrukten (wie z. B. Konditionale), die über den Constraintspeicher synchronisiert werden. Berechnung kann durch sogenannte Threads in nebenläufig ausgeführte Einheiten unterteilt werden. Geeignete Programmierabstraktionen ermöglichen es dem Benutzer, neue Suchstrategien zu programmieren [SS94, Sch97b]. Durch Prozeduren höherer Ordnung können vielfältige Programmierabstraktionen modelliert werden. Oz unterstützt funktionales und objekt-orientiertes Programmieren durch geeignete Syntax und vordefinierte Prozeduren. Dadurch werden kompakte und modulare Programme ermöglicht.

All diese Vorteile lassen Oz als eine geeignete Sprache erscheinen, um kombinatorische Probleme elegant modellieren zu können. Die entscheidende Frage ist jedoch, ob mit Oz viele kombinatorische Probleme auch effizient gelöst werden können. Mit dieser Arbeit können wir diese Frage (zumindest für die untersuchten Probleme) positiv beantworten.

## 1.3 Beitrag

In dieser Arbeit zeigen wir, daß verschiedene kombinatorische Probleme in der nebenläufigen Constraintsprache Oz effizient gelöst werden können. Damit vereint Oz die Vorteile der nebenläufigen Constraintprogrammierung für die Modellierung mit der Effizienz zum Problemlösen, wie man sie von Systemen wie CHIP kennt.

Wir führen ein formales Modell von constraintbasiertem Lösen von kombinatorischen Problemen ein, das unabhängig von einer konkreten Programmiersprache ist, und wir zeigen, wie einige der derzeit besten Schedulingtechniken aus dem Operations Research für Propagierung und Distribuierungsstrategien in dieses Modell integriert werden können. Wir erweitern die nebenläufige Constraintsprache Oz, indem wir Propagierung diesem Modell gemäß einbetten und belegen mit

einer Reihe von Fallstudien für große und schwierige Probleme aus dem Gebiet des Scheduling die Leistungsstärke des entwickelten Systems.

Beiträge wurden geleistet in dem Bereich Modellierung, in dem Anwendungsgebiet Scheduling sowie im Design und in der Implementierung von Programmiersprachen. Die Beiträge im einzelnen sind wie folgt.

### Modellierung von constraintbasiertem Lösen kombinatorischer Probleme

- Wir stellen ein formales Modell von constraintbasiertem Lösen kombinatorischer Probleme vor, das unabhängig von einer konkreten Programmiersprache ist. Das Modell beruht auf Propagierung und Distribuierung und ermöglicht die Integration wichtiger Propagierungskonzepte (z. B. globaler oder redundanter Constraints [AB93, Van89]) wie auch anwendungsspezifischer Distribuierungsstrategien (z. B. für Scheduling [CL94a, BPN95b, Wür96]). (Kapitel 2)

- Wir integrieren in das vorgestellte formale Modell einige der derzeit besten Techniken für disjunktive und kumulative Schedulingprobleme aus dem Operations Research [CP89, AC91, MS96] und der Constraintprogrammierung [CL94a, BPN95b, Nui94] (dies betrifft sowohl Propagierungstechniken (Edge-Finding [AC91]) als auch Distribuierungsstrategien für disjunktive Schedulingprobleme). Außerdem führen wir Erweiterungen für Distribuierungsstrategien ein, die durch das Modell nahegelegt werden (Hinzunahme redundanter Constraints). (Kapitel 5 und 6)

### Design und Implementierung von Programmiersprachen

- Wir integrieren das Modell von Propagierung in das Berechnungsmodell von Oz. Dabei zeigen wir insbesondere auf, wie eine gute Benutzbarkeit und Expressivität unterstützt werden kann und wie die eingeführten Schedulingtechniken besonders berücksichtigt werden. (Kapitel 7)

- Wir geben eine ausführliche Darstellung eines Implementierungsmodells für Propagierung in einer nebenläufigen Constraintsprache an. Insbesondere beschreiben wir eine Reihe von Optimierungen sowie Techniken, um Schedulinganwendungen besonders zu unterstützen. (Kapitel 11)

### Constraintbasiertes Scheduling

- Wir übertragen eine neue Schedulingtechnik [MS96] für disjunktive Schedulingprobleme aus dem Operations Research in die Constraintprogrammierung, verstärken die Propagierung (stärkeres Edge-Finding) und verallgemeinern diese Technik auf kumulative Schedulingprobleme. (Kapitel 5)

- Wir lösen erstmals mit Constraintprogrammierung Probleme aus einem neuen Anwendungsgebiet in der Autoelektronik sowie der Luft- und Raumfahrtindustrie (taktgesteuerte

Echtzeitsysteme). Bei der Lösung sehr großer Probleme kommen die in Oz integrierten Schedulingtechniken zum Einsatz. (Kapitel 10)

- Wir realisieren eine interaktive, grafische Werkbank zur Lösung von disjunktiven und kumulativen Schedulingproblemen, bei der der Benutzer die in Oz verfügbaren Schedulingtechniken beliebig kombinieren kann. Darüberhinaus zeigen wir, daß heuristische Verfahren [CL95] mit Constraintprogrammierung in Oz verbunden werden können. (Kapitel 9)

- Wir nehmen eine umfassende Evaluierung von Schedulingtechniken für Job-Shop-Probleme [GJ79] vor und analysieren die für Scheduling wichtigen Entwurfsentscheidungen bei der Integration in eine Constraintsprache. (Kapitel 9)

## 1.4 Gliederung der Arbeit

Diese Arbeit ist in zwei Teile gegliedert. Im ersten Teil stellen wir das sprachunabhängige formale Modell für Propagierung und Distribuierung vor und zeigen, wie effiziente Schedulingtechniken aus dem Operations Research in das Modell integriert werden können. Im zweiten Teil wird das beschriebene Modell in die Sprache Oz integriert und wir zeigen anhand einer Reihe von Fallstudien die Effizienz des resultierenden Systems.

Kapitel 2 beschreibt ein sprachunabhängiges Berechnungsmodell von Propagierung und Distribuierung. Das Berechnungsmodell wird in Kapitel 3 anhand einiger Beispiele veranschaulicht. Kapitel 4 enthält Beispiele von Constraints und Distribuierungsstrategien, die oft Verwendung finden. Die folgenden zwei Kapitel beschäftigen sich mit Techniken zur Lösung von Schedulingproblemen. In Kapitel 5 werden hierzu spezielle Constraints vorgestellt, deren Propagierung auf Techniken des Operations Research zurückgeht. In Kapitel 6 stellen wir spezielle Distribuierungsstrategien zur Lösung von Schedulingproblemen vor.

Kapitel 7 beschreibt die Integration der in Kapitel 2 vorgestellten Modellierung in das Berechnungsmodell der Sprache Oz. Die Kapitel 8 bis 10 enthalten Fallstudien zur Lösung von kombinatorischen Problemen. In Kapitel 8 wird ein Projekt zur Erstellung eines wöchentlichen Stundenplans für eine Fachhochschule vorgestellt. In Kapitel 9 wird eine Werkbank zur Lösung von Schedulingproblemen beschrieben, die auch zur Evaluierung der in Oz implementierten Schedulingtechniken verwendet wird. Kapitel 10 enthält die Beschreibung eines Prototypen zur Erstellung eines Ablaufplans für taktgesteuerte Echtzeitsysteme. Die Integration von Propagierung in die Implementierung von Oz wird in Kapitel 11 dargestellt. Kapitel 12 beschreibt eine spezielle Propagierungstechnik. Verschiedene andere Systeme zum Lösen kombinatorischer Probleme werden in Kapitel 13 beschrieben und die Effizienz von Oz wird anhand einer Reihe von Beispielen (auch für Scheduling) evaluiert.

# Teil I

# Grundlagen

# Kapitel 2

# Propagierung und Distribuierung

In dieser Arbeit betrachten wir kombinatorische Probleme. Ein kombinatorisches Problem kann durch eine Menge von Variablen und eine Menge von Bedingungen beschrieben werden, die zwischen den Variablen gelten müssen. Für solche Bedingungen verwenden wir den Begriff *Constraints*. Die Variablen dürfen nur ganzzahlige Werte aus einem *endlichen Bereich* (engl. *finite domain*) annehmen. Eine *Lösung* eines kombinatorischen Problems ist eine Belegung jeder Variablen mit einem Wert aus dem ihr zugehörigen Bereich, so daß alle Constraints erfüllt sind. Die Menge aller möglichen Belegungen von Variablen bildet den *Suchraum*. Der Suchraum ist in der Regel bei weitem zu groß, um durch vollständiges Absuchen Lösungen zu finden (so führen schon 10 Variablen mit je 1 000 erlaubten Werten zu einem Suchraum der Größe $10^{30}$).

Den meisten derzeit erfolgreichen Systemen zum constraintbasierten Lösen von kombinatorischen Problemen liegen die Konzepte der *Propagierung* zur Reduktion des Suchraumes und der *Distribuierung* zur Steuerung der Lösungssuche zugrunde [COS96, ILOG96b, CL96a, CD96, ECR96, COC97, Pro97]. In diesem Kapitel führen wir ein formales Modell von constraintbasiertem Lösen kombinatorischer Probleme ein, das unabhängig von einer konkreten Programmiersprache ist. Während das Modell zwar sprachunabhängig ist, kann es doch effiziente Implementierungen beschreiben. So können mit dem entwickelten Kalkül sowohl wichtige Propagierungskonzepte (z. B. globale oder redundante Constraints) als auch anwendungsspezifische Distribuierungsstrategien (z. B. für Scheduling) modelliert werden.

Das hier vorgestellte Modell formalisiert die Begriffe und Konzepte, die bereits in [SSW98] für die Sprache Oz formuliert wurden.

## 2.1 Prinzipielle Ideen

Wir beschreiben ein kombinatorisches Problem durch eine sogenannte *Spezifikation*. Eine Spezifikation enthält in geeigneter Form die Bereiche der Variablen und die Constraints, die das Problem beschreiben.

Die prinzipielle Idee von Constraintprogrammierung ist es, Constraints nicht nur für einen Test von Belegungen zu verwenden, sondern sie aktiv den Suchraum einschränken zu lassen. Hierbei werden Werte aus dem Bereich einer Variablen entfernt, die in keiner Lösung als Belegung

dieser Variablen vorkommen können. Dieses Vorgehen nennt man *Propagierung* oder auch *Constraintpropagierung*. Die stärkste Propagierung erreicht man, indem alle Werte aus Bereichen ausgeschlossen werden, die in keiner Lösung aller Constraints zusammen vorkommen können, sogenannte *globale Konsistenz* wird sichergestellt [Fre88, Dec92]. Doch das Sicherstellen von globaler Konsistenz ist in der Regel sehr zeitaufwendig (Lösen kombinatorischer Probleme ist NP-vollständig [GJ79]).

Wir sind deshalb an einer effizienteren Alternative interessiert, die dennoch den Suchraum möglichst stark verkleinert. Anstatt alle Constraints zusammen, betrachten wir nun einzelne Constraints $c$, die in der Spezifikation vorkommen. Nun werden für $c$ alle Werte aus einem Bereich ausgeschlossen, die garantiert in keiner Lösung von $c$ als Belegung der zugehörigen Variable vorkommen können. Auf diese Weise wird *lokale Konsistenz* für jeden Constraint sichergestellt (in der Literatur ist auch der Begriff Kantenkonsistenz gebräuchlich [Mac77]).

Betrachten wir zum Beispiel die Variablen $X$ und $Y$ mit den Bereichen $\{1,\ldots,100\}$ und den Constraint $X+80 \leq Y$, so kann der Bereich von $X$ zu $\{1,\ldots,20\}$ und der Bereich von $Y$ zu $\{80,\ldots,100\}$ verkleinert werden, ohne daß eine Lösung verloren geht. Dadurch werden von anfangs 10 000 möglichen Belegungen 9 600 ausgeschlossen.

Obwohl auch durch Sicherstellen von lokaler Konsistenz der Suchraum mitunter drastisch verkleinert werden kann, ist globale Konsistenz doch stärker. Lokale Konsistenz reicht allein nicht aus, um Inkonsistenz zu erkennen. Ein Beispiel ist das Problem, daß drei Variablen $X,Y,Z$ paarweise verschiedene Werte annehmen sollen, wobei für jede Variable der Bereich $\{0,1\}$ zugelassen ist. Bei einer Modellierung des Problems durch die drei Constraints $X \neq Y$, $X \neq Z$ und $Y \neq Z$ gilt lokale Konsistenz, da durch keinen Constraint ein Wert ausgeschlossen werden kann. Offensichtlich können aber nicht alle Constraints zusammen erfüllt sein; globale Konsistenz gilt also nicht.

Doch auch das Sicherstellen von lokaler Konsistenz ist oft sehr kostspielig. Im schlimmsten Fall sind alle möglichen Belegungen von Variablen zu betrachten, über die der Constraint definiert ist. Für viele Anwendungen ist es ausreichend, lokale Konsistenz nur zu *approximieren*, das heißt wir schwächen die Propagierung weiter ab. Dazu werden zwar für einen Constraint $c$ Werte aus Bereichen entfernt, die in keiner Lösung von $c$ vorkommen. Dies müssen aber nicht alle möglichen Werte sein, die zur Zusicherung von lokaler Konsistenz entfernt würden. Diesen Berechnungsdienst leistet ein sogenannter *Projektor* für einen Constraint. Eine *Konfiguration* enthält eine Menge von Bereichen und eine Menge von Projektoren.

Während eine Spezifikation ein kombinatorisches Problem hauptsächlich beschreibt, enthält eine Konfiguration die Propagierungsvorschriften (durch Projektoren realisiert), wie der Suchraum den Constraints der Spezifikation entsprechend verkleinert werden soll. Für einen Constraint kann es mehrere Projektoren geben, die sich in der Stärke der Propagierung unterscheiden. Oft geht eine stärkere Propagierung mit einer größeren Laufzeitkomplexität einher. Für die Auswahl eines Projektors zur Problemmodellierung in einer Konfiguration muß deshalb sowohl die mögliche Reduktion des Suchraumes als auch die dafür anfallenden Kosten bzgl. Rechenzeit und Speicherplatz berücksichtigt werden.

Doch weder lokale Konsistenz noch globale Konsistenz reicht in der Regel aus, um Lösungen eines kombinatorischen Problems zu bestimmen. Dazu müssen wir Belegungen aufzählen. Dies sollte jedoch nicht so geschehen, daß für alle Variablen gleichzeitig eine Belegung vorgenommen

wird, da wir dann nur unzureichend von Propagierung profitieren. Stattdessen nehmen wir eine Fallunterscheidung vor. Sei $K$ eine Konfiguration und $v$ eine Variable. Dann erzeugen wir aus $K$ eine Menge von Konfigurationen $K_1$ bis $K_m$, indem zum Beispiel in jedem $K_i$ die Variable $v$ mit einem anderen Wert aus ihrem Bereich belegt wird. Einen solchen Schritt nennen wir *Distribuierungsschritt* und sprechen von *Distribuierung*. Jedes $K_i$ entspricht einem einfacheren Problem als $K$, da bereits eine Variable belegt wurde. Wir wählen nun eine Konfiguration unter $K_1$ bis $K_m$ aus und propagieren wieder, um den Suchraum zu verkleinern (der Suchraum ergibt sich ja aus den möglichen Belegungen bzgl. der in einer Konfiguration enthaltenen Bereiche). Führt ein solches Vorgehen dazu, daß durch Propagierung ein Bereich leer wird, so können die Bereiche der betrachteten Konfiguration zu keiner Lösung des Problems führen. Wir können dann eine andere in einer Fallunterscheidung erzeugte Konfiguration heranziehen und für diese versuchen, Lösungen zu finden. Um möglichst viel Propagierung zu erreichen, sollte ein Distribuierungschritt erst dann vorgenomen werden, wenn keine Propagierung mehr den Suchraum verkleinern kann. In späteren Kapitel werden wir sehen, wie weitaus komplexere Distribuierungschritte als lediglich das Belegen von Variablen vorgenommen werden können.

Durch das Zusammenspiel von *Propagierung und Distribuierung* kann somit ein kombinatorisches Problem gelöst werden. Die Effizienz der Lösungssuche wird entscheidend durch die Modellierung durch geeignete Constraints und Projektoren sowie durch die Wahl einer geeigneten Distribuierungsstrategie beeinflußt.

In diesem Kapitel werden wir die oben erwähnten Konzepte wie Spezifikation, Projektor, Konfiguration oder Distribuierung formalisieren. Wir werden einen Kalkül auf Spezifikationen angeben, mit dem Lösungen von kombinatorischen Problemen berechnet werden können. Diesen Kalkül werden wir für Konfigurationen verfeinern, so daß er als Grundlage einer effizienten Implementierung zum Lösen kombinatorischer Probleme dienen kann.

Abschnitt 2.2 beschreibt, wie ein kombinatorisches Problem durch Spezifikationen beschrieben werden kann. Der nachfolgende Abschnitt enthält einen Kalkül, um Lösungen von Spezifikationen zu berechnen. In Abschnitt 2.4 werden Projektoren eingeführt und in Abschnitt 2.5 wird der vorgestellte Kalkül für Konfigurationen verfeinert. Abschnitt 2.6 geht auf Distribuierung für Konfigurationen ein. Das Kapitel endet mit einigen historischen Bemerkungen.

## 2.2 Spezifikationen

In diesem Abschnitt definieren wir Spezifikationen, mit denen wir Probleme beschreiben.

Zuerst legen wir für alle Spezifikationen eine Menge $V$ von Variablen fest:

$$V = \{x_1, \ldots, x_n\}.$$

Ein *Constraint* ist eine Formel der Prädikatenlogik erster Stufe mit geeigneter Signatur und freien Variablen $y_1, \ldots, y_m$ mit $\{y_1, \ldots, y_m\} \subseteq V$. Wir bezeichnen einen einzelnen Constraint mit $c$. Die Menge $\{y_1, \ldots, y_m\}$ bezeichnen wir als *Argumente* des Constraints $c$ und schreiben dafür $\mathcal{V}(c)$. Ein Constraint $c$ denotiert eine $m$-stellige Relation $rel(c)$ über den ganzen Zahlen. Zum Beispiel denotiert $X < Y$ die Relation $\{(a,b) | a \in \mathbb{Z}, b \in \mathbb{Z}, a < b\}$. Für einen Constraint $c$ bezeichnet $\rho(c)$ die $n$-stellige Erweiterung der $m$-stelligen Relation $rel(c)$ über den ganzen Zahlen. Weitere Beispiele für Constraints sind

$$X = 67 \qquad X = Y$$
$$X^2 - Y^2 = Z^2 \qquad X + Y + Z < U \qquad X + Y \neq 5 \cdot Z$$
$$X_1, \ldots, X_9 \text{ sind paarweise verschieden.}$$

Wir bezeichnen eine Menge von Constraints mit $C$. Die Schreibweise $c(C)$ bezeichnet den Constraint $\bigwedge_{c \in C} c$.

Wir schreiben $c_1 \models c_2$ für $\rho(c_1) \subseteq \rho(c_2)$. So gilt zum Beispiel $X = 5 \models X < 10$ oder $X < Y \models X \leq Y$. Wir schreiben $c_1 \models\!\!\models c_2$, falls $c_1 \models c_2$ und $c_2 \models c_1$ gilt.

Ein Constraint $c_1$ *subsumiert* einen Constraint $c_2$ (engl. *entails*), wenn $c_1 \models c_2$ gilt. Wir sagen dann auch, daß $c_1$ *stärker* ist als $c_2$. Gilt auch noch $c_1 \neq c_2$, so heißt $c_1$ *echt stärker* als $c_2$. Entsprechend heißt $c_2$ *schwächer* (bzw. *echt schwächer*) als $c_1$. Ein Constraint $c_1$ *dissubsumiert* einen Constraint $c_2$ (engl. *disentails*), wenn $c_1$ den Constraint $\neg c_2$ subsumiert. Ein Constraint $c_1$ ist *äquivalent* zu einem Constraint $c_2$, wenn $c_1 \models\!\!\models c_2$ gilt.

Eine *Belegung* $\alpha$ ist eine Funktion von der Menge aller Variablen auf $\mathbb{Z}$. Eine *Lösung* eines Constraints $c$ ist eine Belegung $\alpha$, für die $(\alpha(x_1), \ldots, \alpha(x_n)) \in \rho(c)$ gilt. Wir schreiben hierfür auch $\alpha \models c$. Für eine Belegung $\alpha$ schreiben wir $\{x_1 \mapsto \alpha(x_1), \ldots\}$. In Beispielen führen wir in $\alpha$ nur die Argumente eines Constraints $c$ auf, da für $\alpha \models c$ die Variablen $v \notin \mathcal{V}(c)$ beliebige Werte in $\alpha$ annehmen können. So schreiben wir zum Beispiel $\{X \mapsto 1, Y \mapsto 3\} \models X < Y$.

Eine spezielle Art von Constraints sind *Bereichsconstraints*. Ein Bereichsconstraint ist entweder $\bot$ oder $x_1 \in \delta_1 \wedge \ldots \wedge x_n \in \delta_n$, wobei $\delta_i$, $1 \leq i \leq n$, eine endliche, nicht-leere Teilmenge von $\mathbb{Z}$ ist. $\bot$ denotiert die leere Relation $\emptyset$ und $x_1 \in \delta_1 \wedge \ldots \wedge x_n \in \delta_n$ denotiert die Relation $\{(a_1, \ldots, a_n) | a_i \in \delta_i, 1 \leq i \leq n\}$. Einen Bereichsconstraint bezeichnen wir mit $d$. Die zugehörige Relation bezeichnen wir mit $\rho(d)$. Für einen Bereichsconstraint $d = x_1 \in \delta_1 \wedge \ldots \wedge x_n \in \delta_n$ bezeichnet $\delta_i$ den *Bereich* von $x_i$ und wir schreiben dafür $dom(x_i, d)$. Die Menge aller möglichen Bereichsconstraints wird mit $\mathcal{D}$ bezeichnet. Wir sagen ein Constraint $c$ ist *konsistent* bzgl. eines Bereichsconstraints $d$, falls es für $d \wedge c$ mindestens eine Lösung gibt.

Wir schreiben $m\#n$ für den endlichen Bereich $\{m, \ldots, n\}$. Wir schreiben $[m_1\#n_1 \ldots m_k\#n_k]$ für die Vereinigung der Bereiche $m_i\#n_i, 1 \leq i \leq k$. Die Schreibweise $x = n$ ist eine Abkürzung für $x \in n\#n$. So bezeichnet $[1\#5\ 8\#10]$ die Menge $\{1, \ldots, 5, 8, \ldots, 10\}$. Gilt $\delta \neq \{\min(\delta), \ldots, \max(\delta)\}$, so sagen wir, daß $\delta$ *Lücken* besitzt. Eine Lücke ist ein Wert $v \in \{\min(\delta), \ldots, \max(\delta)\}$, so daß $v \notin \delta$ gilt.

Der Bereichsconstraint $d = x_1 \in \delta_1 \wedge \ldots \wedge x_n \in \delta_n$ *determiniert* die Variable $x_i$, wenn $\delta_i = \{m\}$ für ein geeignetes $m \in \mathbb{Z}$ gilt. Ein Bereichsconstraint $d$ heißt *determiniert*, wenn $\rho(d)$ genau ein Element enthält.

Durch eine Menge von Constraints und einen Bereichsconstraint kann ein kombinatorisches Problem beschrieben werden. Hierzu führen wir eine *Spezifikation* $(d, C)$ ein. Eine Spezifikation $S = (d, C)$ steht für den Constraint $d \wedge c(C)$; wir schreiben hierfür auch $c(S)$. Eine Belegung $\alpha$ heißt *Lösung* von $(d, C)$, falls $\alpha$ Lösung von $d \wedge c(C)$ ist. Wir sagen, daß $(d, C)$ *inkonsistent* ist, falls $(d, C)$ keine Lösung besitzt. Für eine Spezifikation $(d, C)$ werden wir nur die Variablen und Bereiche in $d$ aufschreiben, die tatsächlich auch Argumente von $c(C)$ sind. Für die übrigen Variablen in $V$ können wir beliebige, nicht-leere Bereiche annehmen.

Sind die von Constraints denotierten Relationen entscheidbar, so kann die Lösung einer Spezi-

fikation in endlicher Zeit berechnet werden, da alle Bereiche von Variablen endlich sind. Das Problem, ob eine Spezifikation eine Lösung besitzt ist jedoch NP-hart [GJ79], da Graphfärbeprobleme oder Erfüllbarkeitsprobleme als Spezifikationen formuliert werden können.

Für eine Spezifikation $(d,C)$ können wir effizient entscheiden, ob $d$ eine Lösung hat (also konsistent ist) oder von einem anderen Bereichsconstraint subsumiert wird. Für $C$ hingegen ist der Test, ob alle Constraints zusammen eine Lösung haben (also global konsistent sind) in der Regel nicht effizient realisierbar (siehe oben). Wir werden Konsistenz nur für jeden einzelnen Constraint in $C$ zusichern (also lokale Konsistenz) oder dies aus Effizienzgründen sogar weiter abschwächen.

Das folgende Problem ist ein bekanntes Zahlenpuzzle. Es sind verschiedene Ziffern für die Buchstaben $S, E, N, D, M, O, R, Y$ zu finden, so daß die Gleichung

$$SEND + MORE = MONEY$$

gilt, sowie $S$ und $M$ von Null verschieden sind (keine führenden Nullen).

Wir spezifizieren das Problem, indem wir für jeden Buchstaben eine Variable einführen, die für die mit dem Buchstaben assoziierte Ziffer steht. Der Bereich jeder dieser Variablen ist 0#9. Somit enthält die zugehörige Spezifikation den Bereichsconstraint $d = S \in 0\#9 \wedge \ldots \wedge Y \in 0\#9$. Beachte, daß wir nur die Variablen aufschreiben, die tatsächlich Argumente von Constraints sind. Wir haben die Constraints $c_1 : S \neq 0$ und $c_2 : M \neq 0$, um Nullen als erste Ziffer einer Zahl zu verbieten, und den Constraint $c_3$, daß die Variablen paarweise verschiedene Werte annehmen müssen. Die Gleichung des Rätsels kann durch den Constraint $c_4$:

$$\begin{array}{rrr} & & 1000\cdot S+100\cdot E+10\cdot N+D \\ + & & 1000\cdot M+100\cdot O+10\cdot R+E \\ = & 10000\cdot M+ & 1000\cdot O+100\cdot N+10\cdot E+Y \end{array}$$

beschrieben werden. Eine Spezifikation des Problems ist somit $(d,\{c_1,c_2,c_3,c_4\})$. Die Belegung $\{S\mapsto 9, E\mapsto 5, N\mapsto 6, D\mapsto 7, M\mapsto 1, O\mapsto 0, R\mapsto 8, Y\mapsto 2\}$ ist eine Lösung dieser Spezifikation.

## 2.3 Lösen von Spezifikationen

In diesem Abschnitt stellen wir einen Kalkül (ein Regelsystem) vor, mit dem Lösungen von Spezifikationen durch Propagierung und Distribuierung berechnet werden können. Ziel dieses Abschnitts ist es, grundlegende Ideen zu vermitteln. Deshalb werden wir hier weniger auf Effizienz achten. In Abschnitt 2.5 werden wir dann den Kalkül so verfeinern, daß er als Grundlage einer effizienten Berechnung von Lösungen dienen kann.

Eine Menge von Spezifikationen $M = \{S_1,\ldots,S_m\}$ steht für den Constraint $c(S_1) \vee \ldots \vee c(S_m)$, wofür wir auch $c(M)$ schreiben.

Abbildung 2.1 zeigt die Regeln des Kalküls. Dabei bedeutet $C \uplus \{c\}$ die Vereinigung $C \cup \{c\}$, wobei $c \notin C$ gilt. Die Relation $\rightarrow^*$ ist der transitive und reflexive Abschluß von $\rightarrow$.

Im folgenden erläutern wir die Regeln des Kalküls.

Für eine Anwendung $(d,C) \rightarrow (d',C)$ der Propagierungsregel kann $d'$ echt stärker sein als $d$. Dies bedeutet, daß Bereiche von Variablen in $d'$ echt kleiner sein können als in $d$, was einer

**Abbildung 2.1** Reduktionsregeln für Spezifikationen

**Propagierung** $(d,C) \rightarrow (d',C)$ wenn $d' \models d,\ \exists c \in C:\ d \wedge c \models d'$

**Subsumtion** $(d, C \uplus \{c\}) \rightarrow (d,C)$ wenn $d \models c$

**Dissubsumtion** $(d,C) \rightarrow (\bot,\emptyset)$ wenn $\exists c \in C:\ d \models \neg c$

$\{(\bot,C),S_1,\ldots,S_n\} \rightarrow \{S_1,\ldots,S_n\}$

**Distribuierung** $\{(d,C),\ldots\} \rightarrow \{(d,C \cup C_1),\ldots,(d,C \cup C_m),\ldots\}$

wenn $d \wedge c(C) \models c(C_1) \vee \ldots \vee c(C_m)$

**Struktur** $\dfrac{S \rightarrow S'}{\{S,\ldots\} \rightarrow \{S',\ldots\}}$

Verkleinerung des Suchraums entspricht. Wegen $d \wedge c \models d'$, werden nur Werte aus Bereichen entfernt, die garantiert in keiner Lösung einer Spezifikation vorkommen. Eine Anwendung ist zum Beispiel

$$(X \in 1\#5 \wedge Y \in 1\#5,\ \{X < Y\}) \rightarrow (X \in 1\#4 \wedge Y \in 2\#5,\ \{X < Y\}).$$

Gilt für eine Anwendung $(d,C) \rightarrow (d',C)$, die Beziehung $\rho(d') \subset \rho(d)$, so sprechen wir von *Propagierung*. Beachte, daß in diesem Kalkül ein Constraint sehr viele Reduktionen zuläßt, da $d'$ nicht weiter spezifiziert ist. So ist auch

$$(X \in 1\#5 \wedge Y \in 1\#5,\ \{X < Y\}) \rightarrow (X \in 1\#5 \wedge Y \in 2\#5,\ \{X < Y\})$$

eine gültige Anwendung der Propagierungsregel. Da in der Propagierungsregel immer nur ein einzelner Constraint berücksichtigt wird, stellen wir in diesem Modell lokale Konsistenz sicher. Es wird jedoch deutlich, wie wichtig die Modellierung eines Problems durch eine Spezifikation ist. Verwendet man sehr wenige Constraints (oder nur einen einzigen), so kann das Sicherstellen von lokaler Konsistenz sehr berechnungsintensiv werden (bei einem einzigen Constraint können wir sogar globale Konsistenz sicherstellen). Verwendet man zuviele Constraints, so führt lokale Konsistenz evtl. zu einer sehr geringen Reduktion des Suchraumes. Somit entscheidet die Wahl einer geeigneten Spezifikation mit über die Effizienz der Lösungssuche (siehe auch Abschnitt 2.4 und die folgenden Kapitel).

Mit der Subsumtionsregel können subsumierte Constraints (die keine Propagierung mehr beitragen) aus einer Spezifikation entfernt werden. Ein Beispiel der Regelanwendung ist

$$(X \in 1\#5 \wedge Y \in 6\#9,\ \{X < Y\}) \rightarrow (X \in 1\#5 \wedge Y \in 6\#9,\ \emptyset).$$

Mit den Dissubsumtionsregeln können Spezifikationen entdeckt und eliminiert werden, die keine Lösung besitzen. Ein Beispiel ist

$$(X \in 5\#9 \wedge Y \in 1\#2,\ \{X < Y\}) \rightarrow (\bot,\ \emptyset).$$

Die Distribuierungsregel ermöglicht eine Fallunterscheidung. Dazu wird eine Spezifikation $S$ mit Constraintmengen $C_1$ bis $C_m$ *distribuiert*, wodurch aus $S$ durch Hinzufügen jeweils eines $C_i$ neue

Spezifikationen entstehen. Die Nebenbedingung dieser Regel garantiert, daß durch die Fallunterscheidung keine Lösung verlorengeht. Auf diese Weise kann man ein Problem, für das zum Beispiel keine Propagierung mehr möglich ist, auf Probleme abbilden, für die wieder propagiert werden kann. So ist z. B. für $(X \in 1\#4 \wedge Y \in 2\#5,\ \{X < Y\})$ keine Propagierung möglich. Durch die Regelanwendung

$$\{(X \in 1\#4 \wedge Y \in 2\#5,\ \{X < Y\})\} \ \rightarrow \ \begin{array}{l}\{(X \in 1\#4 \wedge Y \in 2\#5,\ \{X = 1, X < Y\}),\\ (X \in 1\#4 \wedge Y \in 2\#5,\ \{X \neq 1, X < Y\})\}\end{array}$$

gelangt man nach Anwendung der Propagierungsregel auf die erste Spezifikation (für $X = 1$) und zwei Anwendungen der Subsumtionsregel zu der Spezifikation $(X = 1 \wedge Y \in 2\#5,\ \emptyset)$. Für diese Spezifikation sind nun alle Lösungen einfach zu finden; es sind nämlich alle Wertekombinationen aus den Bereichen von $X$ und $Y$.

Die Strukturregel erlaubt es, die Regeln für Spezifikationen auf Mengen von Spezifikationen zu verallgemeinern.

Wir nennen einen Kalkül auf Spezifikationen *korrekt*, falls für $S_1 \rightarrow S_2$ (bzw. Mengen von Spezifikationen $M_1 \rightarrow M_2$) die Beziehung $c(S_1) \models\!\!\models c(S_2)$ (bzw. $c(M_1) \models\!\!\models c(M_2)$) gilt. Damit lassen alle Reduktionsschritte des Kalküls den mit der Spezifikation (bzw. Spezifikationsmenge) assoziierten Constraint invariant; es werden nur Äquivalenztransformationen vorgenommen.

Eine Spezifikation $(d, \emptyset)$ heißt *gelöst*, falls $d \neq \bot$ gilt. Eine Spezifikation $(\bot, \emptyset)$ heißt *fehlgeschlagen*. Wir nennen einen Kalkül auf Spezifikationen *vollständig*, falls es für jede Spezifikation $S$ eine Ableitung $\{S\} \rightarrow^* \{S_1, \ldots, S_m\}$ gibt, so daß alle $S_i,\ 1 \leq i \leq m$, gelöst sind. Hat $S$ keine Lösung, so muß es eine Ableitung $S \rightarrow^* \emptyset$ geben.

Für eine gelöste Spezifikation $S = (d, \emptyset)$ definiert der Bereichsconstraint $d$ eine Lösung $\alpha$ von $S$ durch $\alpha = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ für $(a_1, \ldots, a_n) \in \rho(d)$.

**Satz 1** *Der Kalkül mit den Regeln aus Abbildung 2.1 ist korrekt und vollständig.*

Beweis.

Wir zeigen zuerst Korrektheit.

Für die Propagierungsregel gilt $d \wedge c(C) \models d' \wedge c(C)$ wegen $d \wedge c \models d'$ für ein $c \in C$. Andererseits gilt $d' \wedge c(C) \models d \wedge c(C)$ wegen $d' \models d$. Somit folgt $d' \wedge c(C) \models\!\!\models d \wedge c(C)$.

Für die Subsumtionsregel gilt $d \wedge c(C) \wedge c \models\!\!\models d \wedge c(C)$ direkt wegen $d \models c$.

Für die erste Dissubsumtionsregel gilt $d \wedge c(C) \models\!\!\models \bot \wedge c(\emptyset)$, da $d \wedge c(C) \models\!\!\models \bot$ gilt (wegen $d \models \neg c$ für ein $c \in C$).

Für die zweite Dissubsumtionsregel ist die Korrektheit offensichtlich.

Für die Distribuierungsregel gilt $d \wedge c(C) \models\!\!\models d \wedge c(C) \wedge (c(C_1) \vee \ldots \vee c(C_m)) \models\!\!\models (d \wedge c(C) \wedge c(C_1)) \vee \ldots \vee (d \wedge c(C) \wedge c(C_m))$ wegen der Annahme $d \wedge c(C) \models c(C_1) \vee \ldots c(C_m)$.

Die Korrektheit der Strukturregel folgt sofort aus der Korrektheit der übrigen Regeln.

Die Vollständigkeit des Kalküls kann wie folgt bewiesen werden. Sei $(d, C)$ eine Spezifikation, auf die wir die Distribuierungsregel anwenden:

$$\{(d,C)\} \rightarrow \{(d,C\cup\{c_1\}),\ldots,(d,C\cup\{c_m\})\}$$

Dabei ist $m=|\rho(d)|$ und $c_i$ ist $x_1=a_1\wedge\ldots\wedge x_n=a_n$ für $(a_1,\ldots,a_n)\in\rho(d)$, $1\leq i\leq m$, und alle $c_i$ paarweise verschieden. Offensichtlich gilt $d\wedge c(C)\models c_1\vee\ldots\vee c_m$. Durch jeweiliges Anwenden der Propagierungsregel mit einem $c_i$ kann man die Spezifikationsmenge $\{(d_1,C\cup\{c_1\}),\ldots,(d_m,C\cup\{c_m\})\}$ erhalten, so daß $d_i=c_i, 1\leq i\leq m$, gilt. Sei $C_i=C\cup\{c_i\}$. Für jedes $c\in C_i$ gilt dann entweder $d_i\models c$ oder $d_i\models\neg c$. Hat $(d,C)$ eine Lösung, so gelangen wir durch Anwendungen der Subsumtions- und Dissubsumtionsregeln zu einer Spezifikationsmenge

$$\{(d_1,\emptyset),\ldots,(d_k,\emptyset)\}.$$

Da der Kalkül korrekt ist, definiert jedes der $d_i, 1\leq i\leq k$, eine Lösung von $(d,C)$. Da auch keine Lösungen verloren gegangen sein können, ist jede Lösung von $(d,C)$ eine Lösung eines $(d_i,\emptyset)$. Hat $(d,C)$ keine Lösung, so erhalten wir durch Anwendungen der Dissubsumtionsregel eine leere Spezifikationsmenge. □

Für die Propagierungsregel $(d,C)\rightarrow(d',C)$ ist nicht spezifiziert, um wieviel die Bereiche in $d$ beim Übergang zu $d'$ verkleinert werden. Die folgende Definition charakterisiert für einen Constraint $c$ und einen Bereichsconstraint $d$ den stärksten Bereichsconstraint $\overline{d}$, der von $d\wedge c$ subsumiert wird (vgl. auch [Ben96] und Abschnitt 2.7).

**Definition 1** *Ein Bereichsconstraint $\overline{d}$ heißt* Projektion *von $c$ auf $d$, wenn $d\wedge c\models\overline{d}$ und $\forall d',\ d\wedge c\models d' : \overline{d}\models d'$ gilt.*

So ist zum Beispiel $X\in 1\#3\wedge Y\in[2\ 4\ 6]$ die Projektion von $2\cdot X=Y$ auf $X\in 1\#3\wedge Y\in 0\#7$.

Für die folgende Proposition nehmen wir an, daß der Test, ob eine Belegung eine Lösung eines Constraints $c$ ist, die Zeitkomplexität $O(|\mathcal{V}(c)|)$ hat.

**Proposition 1** *Die Projektion $\overline{d}$ von $c$ auf $d=x_1\in\delta_1,\ldots,x_n\in\delta_n$ kann in $O(m\cdot k^m)$ berechnet werden, wobei $k=\max(\{|\delta_1|,\ldots,|\delta_n|\})$ und $m=|\mathcal{V}(c)|$ gilt.*

Beweis. Sei $\mathcal{V}(c)=\{x_1,\ldots,x_m\}\subseteq V$, sei $\Delta$ die Menge $\delta_1\times\ldots\times\delta_m$. Es gilt $|\Delta|\leq k^m$. Seien $\overline{\delta_i}, 1\leq i\leq m$, leere Mengen. Nun wird für jedes $(a_1,\ldots,a_m)\in\Delta$ geprüft, ob $x_1=a_1\wedge\ldots\wedge x_m=a_m\models c$ gilt. Ist dies der Fall, gilt $\overline{\delta_i}:=\overline{\delta_i}\cup\{a_i\}, 1\leq i\leq m$. Bleibt nach dieser Konstruktion ein $\overline{\delta_i}$ leer, so ist $\overline{d}=\bot$, ansonsten ist $\overline{d}=x_1\in\overline{\delta_1}\wedge\ldots\wedge x_m\in\overline{\delta_m}\wedge x_{m+1}\in\delta_{m+1}\wedge\ldots\wedge x_n\in\delta_n$ Unter der Annahme, daß der Test, ob $x_1=a_1\wedge\ldots\wedge x_m=a_m\models c$ erfüllt ist, die Zeitkomplexität $O(m)$ hat, folgt die Behauptung. □

Beachte, daß die Zeitkomplexität des Tests $x_1=a_1\wedge\ldots\wedge x_m=a_m\models c$ oft als konstant angenommen wird, da nur binäre Constraints betrachtet werden (siehe z. B. [Mac77]). In der Regel können jedoch Constraints beliebige Stelligkeit haben (z. B. Gleichungen; siehe auch Kapitel 4).

Für einige spezielle Constraints läßt sich jedoch die Projektion effizient berechnen. Naheliegend ist dies für Constraints wie die Ungleichung $X\leq Y$, für die nur die Grenzen der jeweiligen Bereiche von $X$ und $Y$ betrachtet werden müssen (siehe auch den sogenannten AC-5 Algorithmus

[VDT92] für weitere Beispiele). Aber auch für komplexere Constraints wie den, der besagt, daß aus $n$ Variablen höchstens $k$ den Wert $v$ annehmen dürfen, gibt es effiziente Algorithmen, um die Projektion zu berechnen (siehe Abschnitt 4.2).

Um Constraints bzgl. eines Bereichsconstraints zu charakterisieren gibt es in der Literatur den Begriff der *Kantenkonsistenz* (engl. *arc-consistency*) [Mac77] oder den äquivalenten Begriff der *Bereichskonsistenz* (engl. *domain-consistency*) [VSD95].

**Definition 2** *Ein Constraint $c$ heißt* bereichskonsistent (kantenkonsistent) *bzgl. $d$ genau dann, wenn für alle $a \in dom(x_i, d)$ gilt, daß $x_i = a \wedge d \wedge c$ eine Lösung besitzt.*

Die folgende Proposition zeigt die Beziehung zwischen Bereichskonsistenz und Projektionen.

**Proposition 2** *Die Projektion von $c$ auf $d$ ist der schwächste Bereichsconstraint $\overline{d}$, so daß $\overline{d} \models d$ und $c$ bereichskonsistent bzgl. $\overline{d}$ ist.*

Beweis. Sei $\overline{d} = x_1 \in \overline{\delta_1} \wedge \ldots \wedge x_n \in \overline{\delta_n}$. Die Behauptung $\overline{d} \models d$ gilt nach der Definition von Projektionen. Wäre $c$ nicht bereichskonsistent bzgl. $\overline{d}$, so gäbe es ein $a \in dom(x_i, \overline{d})$, so daß $x_i = a \wedge \overline{d} \wedge c$ keine Lösung besitzt. Da aber dann für ein $\overline{d}' = x_1 \in \overline{\delta_1} \wedge \ldots \wedge x_i \in \overline{\delta_i} \backslash \{a\} \wedge \ldots \wedge x_n \in \overline{\delta_n}$ auch $d \wedge c \models \overline{d}'$ gilt (wegen $d \wedge c \models \overline{d}$) und $\overline{d}'$ echt stärker als $\overline{d}$ ist, könnte $\overline{d}$ keine Projektion sein. Somit ist $c$ bereichskonsistent bzgl. $\overline{d}$.

Nehmen wir nun an, $\overline{d}$ wäre nicht der schwächste Bereichsconstraint, für den die Bedingungen gelten würden. Dann würde es ein $d'$ geben, das echt schwächer als $\overline{d}$ ist. Somit gibt es mindestens ein $a \in dom(x_i, d')$, das in $dom(x_i, \overline{d})$ nicht mehr vorkommt. Da aber $d'$ auch bereichskonsistent ist, würde $d \wedge c \not\models \overline{d}$ gelten (da $a$ für $x_i$ in einer Lösung von $d \wedge c$ vorkommt, aber aus $\overline{d}$ entfernt wurde). Da dies für eine Projektion aber nicht gelten kann, muß $\overline{d}$ der schwächste Bereichsconstraint sein, für den die Bedingungen aus der Proposition gelten. □

In dem hier vorgestellten Kalkül betrachten wir Mengen von Spezifikationen. In einer Implementierung will man aus Effizienzgründen aber nicht sicherstellen, daß Spezifikationen durch Distribuierung nicht mehrfach eingeführt werden. Deshalb wird eine Implementierung des Kalküls Multimengen anstatt Mengen verwenden. Da eine Übertragung aller Ergebnisse auf Multimengen problemlos ist, wird in dieser Arbeit die gebräuchlichere Mengennotation verwendet.

## 2.4 Projektoren

Im durch Abbildung 2.1 beschriebenen Kalkül kann es bei einer Reduktion $(d,C) \rightarrow (d',C)$ mit einem Constraint $c$ Reduktionen zu verschiedenen $d'$ geben. Um Ergebnisse reproduzieren zu können, sollte für eine Implementierung die Regelanwendung für ein $c$ immer deterministisch sein. Im Kalkül wird außerdem nicht festgelegt, wie stark der neue Basisconstraint $d'$ sein soll. Wählt man für eine Reduktion $(d,C) \rightarrow (d',C)$ für $d'$ immer die Projektion von $c$ auf $d$, so kann dies sehr ineffizient sein (siehe Proposition 1). Deshalb führen wir sogenannte Projektoren ein, die eine Projektion nur *approximieren*. Für einen Constraint kann es durchaus verschiedene Projektoren geben, die sich in ihrer Propagierung unterscheiden. Die Wahl eines geeigneten Projektors entscheidet oft mit über die Effizienz, mit der ein Problem gelöst werden kann. Im

nächsten Abschnitt werden wir dann den im vorangehenden Abschnitt vorgestellten Kalkül durch Projektoren verfeinern.

**Definition 3** *Ein* Projektor *$p$ ist ein Paar $(N_p, E_p)$ aus Funktionen $N_p$ und $E_p$ mit der Signatur*

$$N_p : \mathcal{D} \to \mathcal{D}$$
$$E_p : \mathcal{D} \to \{0,1\}$$

$N_p$ bezeichnet eine *Propagierungsfunktion* (im Englischen oft auch als *narrowing-function* bezeichnet; siehe z. B. [BG96, GH88]) und $E_p$ eine *Subsumtionsfunktion* (engl. *entailment*).

Mit $p$ bezeichnen wir einen einzelnen Projektor. Mit $P$ bezeichnen wir eine Menge von Projektoren und mit $\mathcal{P}$ die Menge aller Projektoren.

**Definition 4** *Ein Projektor $p$* realisiert *einen Constraint $c$, wenn gilt*

| | |
|---|---|
| *1.)* $d \wedge c \models N_p(d)$ | *(Korrektheit)* |
| *2.)* $N_p(d) \models d$ | *(Extensionalität)* |
| *3.) wenn $d$ determiniert und $d \models \neg c$, dann $N_p(d) = \bot$* | *(Adäquatheit)* |
| *4.) wenn $d_1 \models d_2$ dann $N_p(d_1) \models N_p(d_2)$* | *(Monotonie)* |
| *5.)* $N_p(N_p(d)) = N_p(d)$ | *(Idempotenz)* |
| *6.) wenn $E_p(d) = 1$, dann $d \models c$* | *(Korrektheit)* |
| *7.) wenn $d$ determiniert und $d \models c$, dann $E_p(d) = 1$* | *(Adäquatheit)* |
| *8.) wenn $d_1 \models d_2$, dann $E_p(d_1) \geq E_p(d_2)$* | *(Monotonie)* |

Für die Propagierungsfunktion besagt Korrektheit, daß durch Anwendung von $N_p$ keine Lösungen von $d \wedge c$ verloren gehen. Die Propagierungsfunktion $N_p$ beschreibt das *Propagierungsverhalten* eines Projektors $p$. Extensionalität garantiert, daß die Propagierungsfunktion Bereichsconstraints immer nur verstärkt. Adäquatheit stellt sicher, daß die Dissubsumtion $d \models \neg c$ spätestens dann erkannt wird, wenn $d$ determiniert ist. Monotonie garantiert, daß das Ergebnis der Propagierungsfunktion für einen Bereichsconstraint mindestens so stark ist wie das Ergebnis für einen schwächeren Bereichsconstraint. Idempotenz ist keine für die Implementierung eines realistischen Kalküls notwendige Eigenschaft wie die übrigen, sondern für eine effiziente Implementierung wünschenswert (man möchte $N_p$ nicht mehrmals hintereinander anwenden müssen).

Für die Subsumtionsfunktion besagt Korrektheit, daß Subsumtion (durch $E_p(d) = 1$) immer korrekt erkannt wird. Adäquatheit stellt sicher, daß die Subsumtion $d \models c$ spätestens dann erkannt wird, wenn $d$ determiniert ist. Monotonie garantiert, daß ein Constraint für einen Bereichsconstraint als subsumiert erkannt wird, wenn dies auch schon für einen schwächeren Bereichsconstraint gilt.

In dieser Arbeit werden die Projektoren $p$, die einen Constraint $c$ realisieren, Susbsumtion und Dissubsumtion bereits erkennen, wenn alle Variablen $v \in \mathcal{V}(c)$ in einem Bereichsconstraint determiniert sind. Auch wird dieser Projektor in der Propagierungsregel nur Bereiche von Variablen reduzieren, die Argumente von $c$ sind.

Mit $c(p)$ bezeichnen wir den durch $p$ realisierten Constraint. Für eine Menge $P$ bezeichnet $c(P)$ den Constraint $\bigwedge_{p \in P} c(p)$. Die *Argumente* eines Projektors $p$ ist die Menge $\mathcal{V}(c(p))$.

**Proposition 3** *Die durch einen Projektor $p$ bestimmte Relation $\rho(c(p))$ ist eindeutig.*

Beweis. Wäre $\rho(c(p))$ nicht eindeutig, würde $p$ ein $c_1$ und ein $c_2$ realisieren, so daß für einen determinierten Bereichsconstraint $d$ sowohl $d \models c_1$ als auch $d \not\models c_2$ gilt. Wegen der Adäquatheit von $E_p$ gilt aber $E_p(d) = 1$ ($p$ realisiert $c_1$) und wegen der Korrektheit von $E_p$ damit $d \models c_2$, da $p$ auch $c_2$ realisiert. Wegen dieses Widerspruchs muß $\rho(c(p))$ eindeutig sein. □

**Definition 5** *Ein Projektor $p$ für einen Constraint $c$ heißt* propagierungsvollständig *genau dann, wenn $N_p(d)$ die Projektion von $c$ auf $d$ ist. Ein Projektor $p$ heißt* subsumtionsvollständig *genau dann, wenn für alle Bereichsconstraints $d$ gilt, daß aus $d \models c$ auch $E_p(d) = 1$ folgt. Ein Projektor heißt* vollständig, *wenn er sowohl propagierungsvollständig als auch subsumtionsvollständig ist.*

Wie bereits erwähnt, sind Projektoren aus Effizienzgründen häufig nicht vollständig. Als einfaches Beispiel betrachten wir $2 \cdot X = Y$ mit dem Bereichsconstraint $d = X \in 1\#4 \wedge Y \in 1\#10$. Ein vollständiger Projektor $p$ verstärkt den Bereichsconstraint zu $X \in 1\#4 \wedge Y \in [2\ 4\ 6\ 8]$. Ein effizienterer Projektor $p'$ braucht z. B. nur das Minimum und das Maximum von $dom(X,d)$ und $dom(Y,d)$ zu berücksichtigen (siehe auch Abschnitt 4.1). Es gilt dann $N_{p'}(d) = X \in 1\#4 \wedge Y \in 2\#8$. Der Projektor $p'$ ist zwar nicht propagierungsvollständig, aber subsumtionsvollständig, da der Constraint $2 \cdot X = Y$ nur dann von einem Bereichsconstraint $d'$ subsumiert ist, falls $d'$ determiniert ist. Wegen der Adäquatheit von $p'$ gilt aber dann auch $E_{p'}(d') = 1$.

Ein weiteres Beispiel ist der Constraint, daß $n$ Variablen paarweise verschiedene Werte annehmen sollen. Sei $v_i$ der Wert, zu dem eine Variable bereits determiniert wurde. Ein effizienter Projektor entfernt dann $v_i$ aus den Bereichen aller noch nicht determinierten Variablen. Die Propagierungsfunktion hat eine quadratische Komplexität in $n$ (wegen der sicherzustellenden Idempotenz). Ein vollständiger Projektor hat eine Komplexität $O(n^2 \cdot k^2)$, wobei $k$ die maximale Kardinalität eines Bereichs einer der $n$ Variablen ist [Reg94]. Für die meisten Anwendungen reicht der unvollständige Projektor aus. Gleiches gilt für viele andere unvollständige Projektoren, die Constraints nur approximieren (siehe Kapitel 3).

Oft werden für unvollständige Projektoren nur die unteren und oberen Schranken von Bereichen berücksichtigt (insbesondere ist dies im Scheduling der Fall; siehe Kapitel 5). Kombiniert man diese Projektoren mit vollständigen Projektoren, die zum Beispiel unzusammenhängende Bereichen erzeugen, so werden die übrigen Projektoren diese Lücken nicht berücksichtigen. Dies ist zu beachten, soll mit den meist weniger effizienten vollständigen Projektoren mehr Propagierung errreicht werden.

## 2.5 Konfigurationen

In diesem Abschnitt werden wir den Kalkül aus Abschnitt 2.3 verfeinern, indem wir Projektoren zur Berechnung verwenden und die Kalkülregeln so verfeinern, daß sie als Grundlage für eine effiziente Berechnung von Lösungen von Spezifikationen dienen können.

Dazu führen wir sogenannte Konfigurationen ein. Eine *Konfiguration* $K = (d,P)$ besteht aus einem Bereichsconstraint $d$ und einer Menge von Projektoren $P$ und steht für den Constraint $d \wedge c(P)$, wofür wir auch $c(K)$ schreiben. Eine Belegung $\alpha$ heißt Lösung einer Konfiguration

$K$, falls $\alpha$ Lösung von $c(K)$ ist. Analog zu Spezifikationen werden wir für eine Konfiguration $(d,P)$ nur die Variablen und Bereiche in $d$ aufschreiben, die tatsächlich auch Argumente von $c(P)$ sind. Eine Menge von Konfigurationen $M = \{K_1,\ldots,K_m\}$ steht für den Constraint $c(K_1) \vee \ldots \vee c(K_m)$, wofür wir auch $c(M)$ schreiben.

Die folgende Definition verbindet Spezifikationen mit Konfigurationen.

**Definition 6** *Eine Konfiguration $(d,P)$ heißt* zulässig *für eine Spezifikation $(d',C)$ genau dann, wenn*

$$d \wedge c(P) \models\!\!\!| \; d' \wedge c(C)$$

*gilt. In diesem Fall sagen wir, daß die Konfiguration die Spezifikation* realisiert.

Falls die Konfiguration $K$ für die Spezifikation $S$ zulässig ist, sind damit die Lösungen von $K$ auch Lösungen von $S$. Haben wir einen Kalkül, der Lösungen von Konfigurationen berechnet, so können wir damit auch Lösungen von Spezifikationen berechnen. Genau dies ist Gegenstand dieses und des folgenden Abschnitts.

Abbildung 2.2 zeigt die Regeln des Kalküls für Konfigurationen. Die größten Unterschiede zum

**Abbildung 2.2** Reduktionsregeln für Konfigurationen

**Propagierung** $(d,P) \to (N_p(d),P)$ wenn $d \neq \bot,\ p \in P,\ d \neq N_p(d)$

**Subsumtion** $(d, P \uplus \{p\}) \to (d,P)$ wenn $d \neq \bot,\ E_p(d) = 1$

**Fehler** $(\bot,P) \to (\bot,\emptyset)$ wenn $P \neq \emptyset$

$\{(\bot,\emptyset),K_1,\ldots,K_m\} \to \{K_1,\ldots,K_m\}$

**Distribuierung** $\{(d,P),\ldots\} \to \{(d,P\cup P_1),\ldots,(d,P\cup P_m),\ldots\}$
wenn $d \wedge c(P) \models c(P_1) \vee \ldots \vee c(P_m),\ P_i \neq \emptyset,\ \Delta(d) = (P_1,\ldots,P_m)$,
$(d,P)$ weder fehlgeschlagen noch gelöst
und keine andere Regel auf $(d,P)$ anwendbar ist (Stabilität)

**Struktur** $\dfrac{K \to K'}{\{K,\ldots\} \to \{K',\ldots\}}$ für eine Reduktion $K \to K'$

Kalkül aus Abschnitt 2.3 liegen in der Propagierungs- und der Distribuierungsregel. Wir beschreiben die Distribuierungsregel und die restlichen Regeln auf Konfigurationsmengen erst im folgenden Abschnitt, da die Distribuierungsregel eine umfangreichere Diskussion erfordert. Wir assoziieren mit einem Problem genau eine Funktion $\Delta$, eine sogenannte *Distribuierungsstrategie*, über die die Distribuierungsregel parametrisiert ist (siehe Abschnitt 2.6). Wir konzentrieren uns in diesem Abschnitt auf die Propagierungsregel, die Subsumtionsregel und die erste Fehlerregel. Durch diese Trennung gewinnen wir erst einmal ein eingehendes Verständnis von Propagierung durch Projektoren. Die hier betrachteten Regeln nennen wir *Normalformregeln*, da mit ihrer Hilfe eine sogenannte *Normalform* einer Konfiguration berechnet werden kann, auf die keine weitere dieser Regeln anwendbar ist.

Durch die Propagierungsregel $(d,P) \to (d',P)$ wird $d$ so verstärkt, daß aus Bereichen in $d$ Werte entfernt werden, die zu keiner Lösung beitragen können. Im Gegensatz zum Kalkül in Ab-

schnitt 2.3 wird diese Regel jedoch nur angewendet, wenn der resultierende Bereichsconstraint $N_p(d)$ echt stärker als $d$ ist. Wird in der Propagierungsregel für einen Projektor $p$ die Propagierungsfunktion $N_p$ angewendet, so sagen wir auch, daß der *Projektor p angewendet* wird. Die Subsumtionsregel entfernt Projektoren $p$ aus einer Konfiguration, für deren realisierten Constraint $c(p)$ Subsumtion erkannt wurde. Die erste Fehlerregel entfernt alle Projektoren aus einer Konfiguration, wenn klar ist, daß die Konfiguration keine Lösung besitzt.

Im folgenden beschreiben wir einige Eigenschaften von Konfigurationen.

Es ist sehr wohl möglich, daß eine Konfiguration $K$ nur vollständige Projektoren enthält, es aber nicht erkannt wird, daß $c(K)$ keine Lösung besitzt. Ein Beispiel ist die zulässige Konfiguration für

$$(X \in 0\#1 \ \wedge\ Y \in 0\#1 \ \wedge\ Z \in 0\#1,\ \{X \neq Y,\ X \neq Z,\ Y \neq Z\}),$$

in der Ungleichungen durch vollständige Projektoren realisiert werden. Jedoch kann durch Anwendung der Normalformregeln nicht $(\bot, \emptyset)$ abgeleitet werden. Dies zeigt nochmals, daß im allgemeinen die durch Propagierung erreichte lokale Konsistenz nicht ausreicht, um globale Konsistenz zuzusichern (also, daß eine Lösung für alle Constraints zusammen existiert).

Eine Spezifikation kann durch viele zulässige Konfigurationen realisiert werden. Dabei unterscheiden sich die Konfigurationen durch die Menge der ausgewählten Projektoren und vor allem durch die Stärke der jeweiligen Propagierungsfunktionen. Oft ist für einen Projektor $p$ abzuwägen zwischen der Stärke der Propagierung durch $N_p$ und dem Aufwand, um $N_p$ zu berechnen. Die richtige Auswahl der Konfiguration entscheidet mit darüber, wie effizient ein Problem durch Constraintprogrammierung gelöst werden kann (siehe z. B. Kapitel 5 oder Kapitel 9).

Eine Konfiguration $(d, P)$, die eine Spezifikation $(d', C)$ realisiert, kann für einen Constraint $c \in C$ auch mehrere Projektoren in $P$ enthalten. So könnte zum Beispiel der Constraint, daß $n$ Variablen paarweise verschiedene Werte annehmen müssen, durch $n \cdot (n-1)/2$ Projektoren für die Ungleichungen $x \neq y$ in einer Konfiguration realisiert werden. Ein anderes Beispiel ist ein Gleichheitsconstraint $\sum a_i \cdot x_i = 0$ mit $a_i \in \mathbb{Z}$. Eine Konfiguration kann für diesen Constraint zwei vollständige Projektoren für die Constraints $\sum a_i \cdot x_i \leq 0$ und $\sum a_i \cdot x_i \geq 0$ enthalten (für Details siehe Abschnitt 4.1).

Umgekehrt ist es natürlich auch möglich, daß eine Konfiguration nur einen einzelnen Projektor für sehr viele Constraints enthält. So könnte eine Spezifikation $O(n^2)$ Ungleichungen $x \neq y$ enthalten, die realisierende Konfiguration aber nur einen einzelnen Projektor, der das entsprechende Propagierungsverhalten zeigt. Im Prinzip könnte man sogar alle Constraints einer Spezifikation durch einen einzigen Projektor realisieren, was aber in der Regel keine effiziente Problemlösung zuläßt. Für eine kompakte und problemnahe Modellierung, versucht man zu erreichen, daß es für einen Constraint in einer Spezifikation genau einen Projektor in der realisierenden Konfiguration gibt.

In einem späteren Abschnitt werden wir sehen, daß es aber oft nützlich ist, eine Spezifikation $(d', C)$ durch eine Konfiguration $(d, P)$ zu realisieren, so daß es $p \in P$ mit $d \ \wedge\ c(P \backslash \{p\}) \models c(p)$ gibt. Da es bei einer Konfiguration auf die Propagierungsfunktionen der Projektoren ankommt, können solche *redundanten Projektoren* nämlich den Suchraum noch weiter einschränken (siehe Kapitel 3). Solche Projektoren sind zwar bzgl. ihrer logischen Semantik redundant, aber bzgl. ihres Propagierungsverhaltens sehr wichtig. Ihre Gegenstücke für Spezifikationen heißen dementsprechend *redundante Constraints*.

**Proposition 4** *Gilt $K_1 \rightarrow K_2$ für zwei Konfigurationen, so gilt $c(K_1) \models\!\models c(K_2)$. Ist $K_1$ für $S$ zulässig, so ist somit auch $K_2$ für $S$ zulässig.*

Beweis. Wir zeigen, daß für $K_1 \rightarrow K_2$ die Äquivalenz $c(K_1) \models\!\models c(K_2)$ gilt. Daraus folgt dann sofort die zweite Behauptung. Dazu betrachten wir zuerst die Propagierungsregel für ein $p \in P$. Es gilt

$$d \wedge c(P) \models\!\models d \wedge c(p) \wedge c(P) \models N_p(d) \wedge c(P)$$

wegen der Korrektheit von $N_p$. Wegen der Extensionalität von $N_p$ gilt aber auch $N_p(d) \wedge c(P) \models d \wedge c(P)$ und damit die Behauptung für die Propagierungsregel.

Für die erste Fehlerregel ist die Behauptung offensichtlich. Für die Subsumtionsregel gilt

$$d \wedge c(P \uplus \{p\}) \models\!\models d \wedge c(p) \wedge c(P) \models\!\models d \wedge c(P)$$

da aus $E_p(d) = 1$ wegen der Korrektheit von $E_p$ sofort $d \models c(p)$ folgt. □

Diese Proposition zeigt also, daß Normalformregeln Äquivalenztransformationen sind; die Regeln sind also *korrekt*.

Oft ist die Anwendung von Projektoren nicht kommutierend. Für zwei Projektoren $p$ und $q$ muß nicht notwendigerweise $N_p(N_q(d)) = N_q(N_p(d))$ gelten. So gilt zum Beispiel für $d = X \in 1\#4 \wedge Y \in 1\#5$ mit $p$ für $X < Y$ und $q$ für $Y < 4$ (jeweils vollständig):

$$N_p(N_q(d)) = X \in 1\#2 \wedge Y \in 2\#3$$
$$N_q(N_p(d)) = X \in 1\#4 \wedge Y \in 2\#3.$$

Beachte, daß für zwei Projektoren $p$ und $q$ selbst $N_p(N_q(N_p(d))) = N_q(N_p(N_q(d)))$ nicht gelten muß. Ein Gegenbeispiel ist $d = X \in 1\#10 \wedge Y \in 1\#10$ und $p$ für $X < Y$ sowie $q$ für $Y < X$.

Um als Grundlage einer effizienten Berechnung von Problemlösungen zu dienen, muß aber zumindest jede maximale Anwendungsreihenfolge von Propagierungsfunktionen das gleiche Ergebnis liefern (maximal heißt hier, daß auf die zuletzt erhaltene Konfiguration keine Propagierungsregel mehr angewendet werden kann). Ansonsten könnte die Effizienz der Lösungssuche von der Anwendungsreihenfolge der Propagierungsregel abhängen. Wir fordern also *Konfluenz* der Regeln (nicht allein der Propagierungsregel). Konfluenz bedeutet, daß wenn von einer Konfiguration $K$ zwei Konfigurationen $K_1$ und $K_2$ durch $K \rightarrow^* K_1$ und $K \rightarrow^* K_2$ erreicht werden, es eine Konfiguration $K'$ gibt, für die $K_1 \rightarrow^* K'$ und $K_2 \rightarrow^* K'$ gilt. Zusätzlich zur Konfluenz sollte es keine unendliche Folge von Regelanwendungen für eine Konfiguration geben (wir fordern also Terminierung).

**Satz 2** *Die Anwendung der Normalformregeln auf eine Konfiguration $(d,P)$ ist terminierend und konfluent.*

Beweis. Wir zeigen zuerst Terminierung. Dazu betrachten wir den Wert von $|\rho(d)| + |P|$, welcher offensichtlich endlich ist. Da jede Anwendung einer Normalformregel $|\rho(d)| + |P|$ verkleinert, folgt die Behauptung (beachte, daß für die Propagierungsregel $N_p(d)$ echt stärker als $d$ ist).

Im Falle der Terminierung reicht es aus, lokale Konfluenz zu beweisen, da daraus Konfluenz folgt [Hue80]. *Lokale Konfluenz* gilt, falls von einer Konfiguration $K$ zwei Konfigurationen $K_1$ und $K_2$

durch $K \rightarrow K_1$ und $K \rightarrow K_2$ erreicht werden, und es eine Konfiguration $K'$ mit $K_1 \rightarrow^* K'$ und $K_2 \rightarrow^* K'$ gibt. Wir beweisen lokale Konfluenz mit einer Fallunterscheidung für die Normalformregeln. Dazu sei $K = (d, P)$. Beachte, daß für zwei Bereichsconstraints $d$ und $d'$ die Äquivalenz $d \models\!\!\models d'$ genau dann gilt, wenn $d = d'$ gilt (aufgrund der Definition von Bereichsconstraints).

1. Gehe $K_1$ bzw. $K_2$ aus $K$ durch Anwendung der Propagierungsregel mit dem Projektor $p_1$ bzw. $p_2$ hervor ($p_1 \in P$ und $p_2 \in P$). Wir nehmen hier $d \neq \bot$ an, da sonst keine Propagierungsregel anwendbar wäre. Wir beweisen die Behauptung, indem wir zeigen, daß jede maximale Hintereinanderausführung von $N_{p_1}$ und $N_{p_2}$ in $K \rightarrow^* K'$ und $K \rightarrow^* K''$ zur gleichen Konfiguration führt (maximal heißt, daß auf $K'$ und $K''$ keine Propagierungsregel mit $N_{p_1}$ oder $N_{p_2}$ mehr angewendet werden kann). Sei $N'_{\tilde{p}}$ (bzw. $N''_{\tilde{p}}$) die Funktion, die den Hintereinanderausführungen von $N_{p_1}$ und $N_{p_2}$ entspricht, die zu $K' = (d', P)$ (bzw. $K'' = (d'', P)$) geführt haben. Sei weiterhin $K = (d, P)$. Wegen der Extensionalität von $N_{p_1}$ und $N_{p_2}$ folgt, daß $N'_{\tilde{p}}$ extensional ist und damit gilt $d' \models d$. Durch Induktion kann gezeigt werden, daß die Funktion $N''_{\tilde{p}}$ monoton ist. Damit folgt aus $d' \models d$ aber sofort $N''_{\tilde{p}}(d') \models N''_{\tilde{p}}(d)$. Da $(d', P)$ aber wegen der Maximalität der Ausführungen nicht weiter reduziert werden kann und $N''_{\tilde{p}}(d) = d''$ gilt, folgt $d' \models d''$. Mit einer analogen Argumentation kann $d'' \models d'$ gezeigt werden und somit gilt insgesamt $d' \models\!\!\models d''$ und damit $d' = d''$. Damit ist lokale Konfluenz für diesen Fall gezeigt.

2. Eine Propagierungsregel und die erste Fehlerregel können nicht gleichzeitig angewendet werden, da in einem Fall $d \neq \bot$ und im anderen $d = \bot$ gelten muß.

3. Wie im vorherigen Fall kann auch nicht gleichzeitig die Subsumtions- und die erste Fehlerregel angewendet werden.

4. Im letzten Fall haben wir die Anwendung einer Propagierungsregel und der Subsumtionsregel zu betrachten. Gelte $(d, P \uplus \{p\}) \rightarrow (d_1, P \uplus \{p\})$ für die Reduktion mit dem Projektor $p_1$ und $(d, P \uplus \{p\}) \rightarrow (d, P)$ für die Anwendung der Subsumtionsregel. Es gilt $d \neq \bot$.

   (a) Wir betrachten zuerst den Fall $d_1 = \bot$, also $N_{p_1}(d) = \bot$. In diesem Fall kann von $(d_1, P \uplus \{p\})$ mit der Fehlerregel zu $(\bot, \emptyset)$ übergegangen werden.

      i. Gilt $p_1 \neq p$, so gilt $(d, P) \rightarrow (d_1, P) \rightarrow (\bot, \emptyset)$.

      ii. Würde $p_1 = p$ gelten, so würde $E_{p_1}(d) = 1$ und damit $d \models c(p_1)$ gelten. Wegen $N_{p_1}(d) = \bot$ gilt aber auch $d \wedge c(p_1) \models \bot$. Somit kann $p_1 = p$ nicht gelten.

   (b) Nun betrachten wir den Fall $d_1 \neq \bot$.

      i. Gilt $p_1 \neq p$, so kann aus $(d, P)$ mit der Propagierungsregel für $p_1$ zu $(d_1, P)$ reduziert werden. Wir haben nun zu zeigen, daß auch $(d_1, P \uplus \{p\}) \rightarrow (d_1, P)$ möglich ist. Wegen $d_1 \models d$ gilt $E_p(d_1) \geq E_p(d)$ wegen der Monotonie von $E_p$. Wegen $E_p(d) = 1$ gilt somit $E_p(d_1) = 1$ und die Behauptung ist bewiesen.

      ii. Würde $p_1 = p$ gelten, dann würde wegen $E_{p_1}(d) = 1$ auch $d \models c(p)$ gelten. Wegen der Monotonie von $N_{p_1}$ gilt aber $d \wedge c(p) \models d_1$ und damit $d \wedge c(p) \models\!\!\models d \models d_1$. Wegen der Extensionalität von $N_{p_1}$ gilt aber auch $d_1 \models d$ und somit $d_1 \models\!\!\models d$. Dies kann aber nicht sein, da dann die Propagierungsregel nicht anwendbar wäre. Somit kann $p_1 = p$ nicht gelten.

□

Die folgende Definition charakterisiert Konfigurationen, auf die keine Normalformregel mehr angewendet werden kann.

**Definition 7** *Eine Konfiguration $K'$ heißt* Normalform *von $K$ genau dann, wenn $K \rightarrow^* K'$ gilt und es kein $K''$ mit $K' \rightarrow K''$ gibt.*

**Korollar 1** *Die Normalformen einer Konfigurationen sind eindeutig.*

Beweis. Folgt sofort aus Satz 2. □

Die folgende Proposition zeigt, daß Normalformen den stärksten Bereichsconstraint charakterisieren, der von einer Konfiguration durch Propagierung abgeleitet werden kann.

**Proposition 5** *Gilt $(d,P) \rightarrow^* (d',P')$ und ist $(d'',P'')$ eine Normalform von $(d,P)$, gilt $d'' \models d'$.*

Beweis. Betrachten wir $K = (d,P)$ und ein $\overline{K} = (\overline{d},\overline{P})$ mit $K \rightarrow^* \overline{K}$, so gilt $\overline{d} \models d$ wegen der Extensionalität jeder Propagierungsfunktion $N_p$, die zu $\overline{K}$ führt. Nun gibt es eine Normalform $(d''',P''')$, so daß $(d,P) \rightarrow^* (d',P') \rightarrow^* (d''',P''')$ gilt. Damit gilt $d''' \models d'$. Da aber wegen Satz 2 auch $d''' = d''$ gilt, folgt somit auch $d'' \models d'$. □

Nun betrachten wir ein Beispiel, um die Berechnung von Normalformen zu veranschaulichen. Dazu sei die Spezifikation $(d, \{2\cdot Y \leq Z, X < Y, Z < 7\}$ mit $d = X \in 1\#10 \wedge Y \in 1\#10 \wedge Z \in 1\#10$ gegeben, die durch die Konfiguration $(d, \{p_{2\cdot Y \leq Z}, p_{X<Y}, P_{Z<7}\}$ realisiert wird. Dabei ist jeder Projektor $p_c$ vollständig für den Constraint $c$.

Wird zuerst $p_{2\cdot Y \leq Z}$ angewendet, so ergibt sich

$$X \in 1\#10 \wedge Y \in 1\#5 \wedge Z \in 2\#10.$$

Eine Anwendung von $p_{X<Y}$ liefert nun

$$X \in 1\#4 \wedge Y \in 2\#5 \wedge Z \in 2\#10.$$

Eine erneute Anwendung von $p_{2\cdot Y \leq Z}$ kann dann diesen Bereichsconstraint zu

$$X \in 1\#4 \wedge Y \in 2\#5 \wedge Z \in 4\#10$$

verstärken. Die Anwendung von $p_{Z<7}$ resultiert in

$$X \in 1\#4 \wedge Y \in 2\#5 \wedge Z \in 4\#6,$$

so daß $Z < 7$ nun vom aktuellen Bereichsconstraint subsumiert ist und der Projektor $p_{Z<7}$ aus der Konfiguration entfernt werden kann. Eine Anwendung von $p_{2\cdot Y \leq Z}$ und $p_{X<Y}$ (in dieser Reihenfolge) ergibt

$$X \in 1\#2 \wedge Y \in 2\#3 \wedge Z \in 4\#6.$$

Nun kann keine weitere Normalformregel mehr angewendet werden.

Jetzt betrachten wir eine andere Anwendungsreihenfolge. Die Anwendung von $p_{Z<7}$ auf $d$ liefert

$$X \in 1\#10 \ \wedge \ Y \in 1\#10 \ \wedge \ Z \in 1\#6.$$

Der Projektor $p_{Z<7}$ kann somit durch die Subsumtionsregel aus der Konfiguration entfernt werden. Die Anwendung von $p_{X<Y}$ verstärkt den Bereichsconstraint zu

$$X \in 1\#9 \ \wedge \ Y \in 2\#10 \ \wedge \ Z \in 1\#6.$$

Die Anwendung von $p_{2\cdot Y\leq Z}$ führt nun zu

$$X \in 1\#9 \ \wedge \ Y \in 2\#3 \ \wedge \ Z \in 4\#6.$$

Die erneute Anwendung von $p_{X<Y}$ verstärkt diesen Bereichsconstraint zu

$$X \in 1\#2 \ \wedge \ Y \in 2\#3 \ \wedge \ Z \in 4\#6.$$

Somit ergeben sich die gleichen Bereiche wie bei der ersten Anwendungsreihenfolge.

Wie die Normalform einer Konfiguration berechnet werden kann, zeigt Algorithmus 2.1. Dieser Algorithmus ist eine Verallgemeinerung des sogenannten AC-3 Algorithmus [Mac77] auf Constraints mit beliebig vielen Argumenten. Dazu betrachten wir eine Konfiguration $(d,P)$. Sei $|P| = e$ und $l = |\mathcal{V}(c(P))|$ die Zahl der Argumente von $c(P)$. Sei $m = \max(\{|\mathcal{V}(c(p))| \mid p \in P\})$ die maximale Zahl von Argumenten eines $c(p)$ und sei $k = \max(\{|dom(x,d)| \mid x \in \mathcal{V}(c(P))\})$ die maximale Kardinalität eines Bereichs in $d$.

**Proposition 6** *Algorithmus 2.1 berechnet für eine Konfiguration $(d,P)$ die Normalform. Die Laufzeitkomplexität des Algorithmus ist $O(e \cdot l \cdot k \cdot (e \cdot m + m \cdot k^m))$.*

Beweis. Der Algoritmus setzt die Subsumtions- und die erste Fehlerregel aus Abbildung 2.2 direkt um. Da $\mathsf{S}$ initial alle Projektoren in $P$ enthält, werden alle $p \in P$ mindestens einmal angewendet. In der Schleife werden dann für $\mathsf{S}$ nur diejenigen Projektoren $p_j$ verwendet, für die der Bereich eines ihrer Argumente verkleinert wurde. Nur solche Projektoren können überhaupt zu Propagierung beitragen. Beachte, daß wir bei der Erweiterung von $\mathsf{S}$ den Projektor $\mathsf{p}$ nur dann nicht berücksichtigen müssen, wenn $\mathsf{p}$ idempotent ist. Für die Laufzeitkomplexität gilt folgendes. Die Bedingung $\mathsf{d} \neq \mathsf{d'}$ kann höchstens $O(k \cdot l)$ mal erfüllt sein, da nur so viele Werte aus den Bereichen entfernt werden können. Ist jeder Projektor vollständig, so muß jeweils für $N_{\mathrm{p}}(\mathsf{d})$ die Projektion berechnet werden, was die Zeitkomplexität $O(m \cdot k^m)$ hat (siehe Proposition 1). Um $\mathsf{S}$ zu vergrößern, müssen jeweils $O(e \cdot m)$ Tests durchgeführt werden. Da außerdem $|S| \leq e$ gilt, ergibt sich insgesamt die Komplexität $O(e \cdot l \cdot k \cdot (e \cdot m + m \cdot k^m))$. □

Die folgende Proposition zeigt, daß wir durch die Normalformregeln feststellen können, ob für eine Spezifikation $(d,C)$ mit einem determinierten Bereichsconstraint $d$ dieser Bereichsconstraint eine Lösung von $c(C)$ ist.

**Proposition 7** *Sei $d$ ein determinierter Bereichsconstraint und sei $(d,P)$ eine zulässige Konfiguration für $(d,C)$. Dann gilt*

$d \models c(C)$ *genau dann, wenn* $(d,P) \rightarrow^* (d,\emptyset)$
$d \models \neg c(C)$ *genau dann, wenn* $(d,P) \rightarrow^* (\bot,\emptyset)$

**Algorithmus 2.1** Berechnung der Normalform einer Konfiguration $(d,P)$

```
S := P;
while S ≠ ∅ do
  if d = ⊥ then return (⊥,∅);
  p := waehle ein p_i ∈ S;
  S := S \ {p};
  if E_p(d) = 1 then P := P \ {p};
  else
    d' := N_p(d);
    if d' ≠ d then
      if d' ≠ ⊥ then
        S := S ∪ { p_j ∈ P | p_j ≠ p, ∃x_i ∈ V(c(p_j)): dom(x_i,d) ≠ dom(x_i,d')};
      end;
    end;
    d := d';
  end;
end;
return (d,P);
```

Beweis.

1. Gelte $d \models c(C)$. Würde $E_{p_i}(d) = 0$ für ein $p_i \in P$ gelten, so würde $d \models \neg c(p_i)$ (wegen der Adäquatheit von $E_{p_i}$) und damit $d \not\models d \wedge c(P) \models\!\!\!\models d \wedge c(C)$ gelten. Somit gilt $E_{p_i}(d) = 1$ für alle $p_i$ und die Behauptung folgt aus Anwendungen der Subsumtionsregel.

2. Gilt $(d,P) \rightarrow^* (d,\emptyset)$, so gilt $E_{p_i}(d) = 1$ für alle $p_i \in P$ (da $d \neq \bot$). Damit folgt $d \models c(p_i)$ für alle $p_i$ und somit $d \models d \wedge c(P) \models\!\!\!\models d \wedge c(C) \models c(C)$.

3. Gelte $d \models \neg c(C)$. Würde $d \models c(p_i)$ für alle $p_i \in P$ gelten, so würde auch $d \models d \wedge c(P) \models\!\!\!\models d \wedge c(C) \models c(C)$ gelten. Somit gibt es ein $p_i \in P$, so daß $d \models \neg c(p_i)$ gilt. Damit folgt aus der Adäquatheit von $N_{p_i}$, daß $N_{p_i}(d) = \bot$ gilt. Durch anschließende Anwendung der ersten Fehlerregel folgt die Behauptung.

4. Gilt $(d,P) \rightarrow^* (\bot,\emptyset)$, so gibt es ein $p_i \in P$ mit $d \wedge c(p_i) \models \bot$ (Korrektheit von $N_{p_i}$). Damit gilt aber $d \models \neg c(p_i)$. Würde $d \models c(C)$ gelten, würde aber auch $d \models d \wedge c(C) \models\!\!\!\models d \wedge c(P)$ gelten. Dies kann aber wegen $d \models \neg c(p_i)$ nicht sein und $d \models \neg c(C)$ muß gelten.

□

Ähnlich wie für Spezifikationen definiert eine Konfiguration $(d,\emptyset)$ Belegungen $\alpha = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ für $(a_1,\ldots,a_n) \in \rho(d)$. Realisiert eine Konfiguration $K$ eine Spezifikation $S$ und gilt $K \rightarrow^* (d,\emptyset)$, so sind alle Belegungen, die $(d,\emptyset)$ definiert, Lösungen von $K$ und damit auch von $S$. Gilt aber $K \rightarrow^* (\bot,\emptyset)$, so kann $S$ keine Lösung besitzen. Oft ist es jedoch nicht möglich, von einer Konfiguration zu $(d,\emptyset)$ mit $d \neq \bot$ oder zu $(\bot,\emptyset)$ zu gelangen. Im nächsten Abschnitt werden wir die Distribuierungsregel verwenden, um Lösungen von Konfigurationen zu berechnen. Hierzu charakterisieren wir zuerst eine Konfiguration $(d,P)$ wie folgt.

**Definition 8** *Eine Konfiguration $K = (d,P)$ heißt*

| | |
|---|---|
| gelöst, | *wenn* $d \neq \bot$ *und* $P = \emptyset$ |
| fehlgeschlagen, | *wenn* $d = \bot$ *und* $P = \emptyset$ |
| stabil, | *wenn* $K$ *in Normalform und weder fehlgeschlagen noch gelöst ist.* |

Wir fassen nun noch einmal das bisherige Vorgehen zur Lösung eines kombinatorischen Problems zusammen. Zuerst beschreibt man das Problem durch eine Spezifikation $(d,C)$. Für diese Spezifikation definiert man dann eine zulässige Konfiguration $(d,P)$, indem geeignete Projektoren ausgewählt werden. Anschließend wenden wir die Normalformregeln auf $(d,P)$ an, bis wir eine Normalform berechnet haben.

**Definition 9** *Sei* $S = (d,C)$ *eine Spezifikation für ein Problem, sei* $(d,P)$ *eine für* $S$ *zulässige Konfiguration und sei* $Y = \mathcal{V}(c(C)) = \{x_1,\ldots,x_{|\mathcal{V}(c(C))|}\}$. *Sei weiterhin* $d' = x_1 \in dom(x_1,d) \wedge \ldots \wedge x_{|Y|} \in dom(x_{|Y|},d)$. *Dann heißt das Quadrupel* $(Y,d',C,P)$ *ein* Modell *für das Problem.*

In Kapitel 3 werden wir für einige Beispiele Modelle definieren.

## 2.6 Distribuierung

In diesem Abschnitt diskutieren wir die Reduktionsregeln aus Abbildung 2.2, die auf Konfigurationsmengen angewendet werden. Dies ist insbesondere die Distribuierungsregel, die uns das Berechnen von Lösungen beliebiger Konfigurationen ermöglicht.

Die Fehlerregel auf Konfigurationsmengen entfernt Konfigurationen, die keine Lösung haben. Die Distribuierungsregel erlaubt es, wie für den Kalkül aus Abschnitt 2.3 (mit Constraints), eine Fallunterscheidung vorzunehmen. Dazu kann eine Konfiguration $K$ auf mehrere Konfigurationen abgebildet werden, die durch Hinzufügen von Projektoren zu $K$ entstanden sind. Ein solches Vorgehen nennen wir *distribuieren*. Durch die hinzugefügten Projektoren kann stärkere Propagierung ermöglicht werden, die den Suchraum weiter verkleinert. Die Funktion $\Delta$ (eine Distribuierungsstrategie), über die die Regel parametrisiert ist (siehe Definition 10), definiert für eine Distribuierungsregel, mit welchen Projektoren distribuiert wird. Beachte aber, daß Distribuierung nicht für fehlgeschlagene oder gelöste Konfigurationen erlaubt ist. Eine Distribuierung ist hier nämlich nicht interessant, da offensichtlich gar keine Lösung existiert, oder aber alle Lösungen bereits leicht berechnet werden können (siehe den vorangehenden Abschnitt). Eine Anwendung der Distribuierungsregel bezeichnen wir auch als *Distribuierungsschritt*. Die Strukturregel erlaubt es, Reduktionsregeln für Konfigurationen auf Konfigurationsmengen zu verallgemeinern.

Beachte, daß die Distribuierungsregel nur für eine Konfiguration $K$ angewendet wird, wenn keine andere Reduktionsregel auf $K$ mehr angewendet werden kann, $K$ also stabil ist. Für dieses Vorgehen in der Distribuierungsregel gibt es zwei Gründe. Zum einen soll der Suchraum so stark wie möglich durch effiziente Propagierung verkleinert werden, bevor wieder ein Suchschritt vorgenommen wird. Zum anderen soll für die Distribuierung so viel Information wie möglich für eine fundierte Entscheidung zur Verfügung stehen (siehe auch den nächsten Paragraphen und speziell Kapitel 6). Ein solches Vorgehen ist in der Constraintprogrammierung üblich (siehe auch Abschnitt 2.7).

Wir sagen, daß eine Regel $M_1 \rightarrow M_2$ auf Konfigurationsmengen *korrekt* ist, falls $c(M_1) \models\!\models c(M_2)$ gilt.

**Proposition 8** *Die Regeln auf Konfigurationsmengen in Abbildung 2.2 sind korrekt.*

Beweis. Da $c(\bot,\emptyset) \models\!\models \bot$ gilt, ist die Korrektheit der zweiten Fehlerregel offensichtlich.

Die Korrektheit der Distribuierungsregel folgt aus $d \wedge c(P) \models\!\models d \wedge c(P) \wedge (c(P_1) \vee \ldots c(P_m)) \models\!\models d \wedge c(P) \wedge c(P_1) \vee \ldots \vee d \wedge c(P) \wedge c(P_m)$ wegen der Annahme $d \wedge c(P) \models c(P_1) \vee \ldots c(P_m)$.

Die Korrektheit der Strukturregel folgt direkt aus der Korrektheit der Reduktionsregeln für Konfigurationen (siehe Proposition 4). □

Der folgende Satz zeigt, daß es mit den in Abbildung 2.2 definierten Regeln möglich ist, alle Lösungen einer Spezifikation zu berechnen, für die eine zulässige Konfiguration gegeben ist. Effiziente und einfache Verfahren, um Lösungen von Konfigurationen (und damit von Spezifikationen) zu finden, werden zum Beispiel in Kapitel 3 präsentiert.

Wir sagen, daß aus einer zulässigen Konfiguration $(d',P)$ für eine Spezifikation $(d,C)$ eine *Lösung* $\alpha$ für $(d,C)$ *berechnet* werden kann, wenn es ein $(d'',\emptyset)$ mit $\{(d',P)\} \rightarrow^* \{(d'',\emptyset),\ldots\}$ gibt, so daß für ein $\alpha = \{x_1 \mapsto a_1,\ldots,x_n \mapsto a_n\}$ mit $(a_1,\ldots,a_n) \in \rho(d'')$ die Subsumtion $\alpha \models d \wedge c(C)$ gilt.

**Satz 3** *Sei $(d',P)$ eine zulässige Konfiguration für eine Spezifikation $(d,C)$. Dann können durch Anwendung der Regeln aus Abbildung 2.2 auf $\{(d',P)\}$ mit einem geeigneten $\Delta$ alle Lösungen von $(d,C)$ berechnet werden.*

Beweis. Wir wenden auf $\{(d',P)\}$ die Distribuierungsregel an:

$$\{(d',P)\} \rightarrow \{(d',P\cup\{p_1\}),\ldots,(d',P\cup\{p_m\})\}$$

Dabei ist $\Delta(d') = (\{p_1\},\ldots,\{p_m\})$. Es gilt $m = |\rho(d')|$ und $c_i$ ist $x_1 = a_1 \wedge \ldots \wedge x_n = a_n$ für $(a_1,\ldots,a_n) \in \rho(d')$, $1 \leq i \leq m$, und alle $c_i$ paarweise verschieden. Der Projektor $p_i$ realisiere den Constraint $c_i$ vollständig. Durch jeweilige Anwendung der Propagierungsregel mit einem $p_i$ erhält man die Konfigurationsmenge $\{(d_1,P\cup\{p_1\}),\ldots,(d_m,P\cup\{p_m\})\}$, so daß $d_i = c_i, 1 \leq i \leq m$, gilt. Sei $P_i = P\cup\{p_i\}$. Für jedes $p \in P_i$ gilt dann entweder $d_i \models c(p)$ oder $d_i \models \neg c(p)$. Hat $(d',P)$ eine Lösung, so gelangen wir durch Anwendungen der Subsumtionsregel und der Fehlerregeln zu einer Konfigurationsmenge

$$\{(d_1,\emptyset),\ldots,(d_k,\emptyset)\},$$

wobei jedes $(d_i,\emptyset), 1 \leq i \leq k$, gelöst ist. Da $(d',P)$ zulässig für $(d,C)$ ist und die Reduktionsregeln alle korrekt sind, folgt sofort, daß eine Lösung von einem $(d_i,\emptyset)$ eine Lösung von $(d,C)$ ist. Da auch keine Lösung verloren geht, ist jede Lösung von $(d,C)$ durch ein $(d_i,\emptyset)$ definiert. Dabei definiert jedes $d_i$ genau eine Lösung $\alpha_i = \{x_1 \mapsto dom(x_1,d_i),\ldots,x_n \mapsto dom(x_n,d_i)\}$. Hat $(d',P)$ keine Lösung, so wird durch Anwendungen der Fehlerregel die leere Konfigurationsmenge abgeleitet. □

**Distribuierungsstrategien**

Nun betrachten wir sogenannte *Distribuierungsstrategien*, die die Distribuierung kontrollieren (ähnlich wie ein Projektor, der die Propagierung eines Constraints kontrolliert). Zu einem Problem assoziieren wir mit einem Modell auch genau eine Distribuierungsstrategie $\Delta$. Die Distribuierungsregel ist über diese Distribuierungsstrategie parametrisiert. Zusammen mit dem Modell für ein Problem bestimmt eine Distribuierungsstrategie, wie effizient ein Problem gelöst werden kann.

**Definition 10** *Eine* $m$-stellige Distribuierungsstrategie $\Delta$ *ist eine Funktion mit der Signatur* $\mathcal{D} \rightarrow \mathcal{P}^m$ *mit* $m \geq 2$. *Gilt* $\Delta(d) = (P_1, \ldots, P_m)$ *und wird die Distribuierungsregel angewendet, sagen wir, daß die Konfiguration* $(d, P)$ *mit* $(P_1, \ldots, P_m)$ distribuiert *wird.*

Wir distribuieren mit einem Tupel $(P_1, \ldots, P_m)$ und nicht mit einer Menge $\{P_1, \ldots, P_m\}$, da die Ordnung der Projektormengen im Tupel für die Suchstrategie benötigt wird, mit der Konfigurationen zur Distribuierung ausgewählt werden.

Wird $(d, P)$ mit $(P_1, \ldots, P_m)$ distribuiert, sagen wir oft auch, daß $(d, P)$ mit $P_i$ distribuiert wird, wenn dann $(d, P \cup P_i)$ näher betrachtet werden soll. Sind die Projektoren, mit denen distribuiert wird, vollständig, so schreiben wir nur die Constraints auf. Gilt also $\Delta(d) = (P_1, \ldots, P_m)$, so schreiben wir $\Delta(P) = (\bigcup_{p \in P_1} c(p), \ldots, \bigcup_{p \in P_m} c(p))$.

Um die Lösungssuche bei Problemen aus der Praxis möglichst gut zu kontrollieren, kann man oft anwendungsspezifisches Wissen zur Entwicklung von Distribuierungsstrategien ausnutzen (siehe z. B. Abschnitt 3.2 und Kapitel 6). Im folgenden stellen wir einige Distribuierungsstrategien vor. Eine sehr einfache Distribuierungsstrategie $\Delta_1$ zählt für alle noch nicht determinierten Variablen (von $x_1$ beginnend) ihre möglichen Werte auf. So ist z. B.

$$\Delta_1(x_1 \in 1\#10 \wedge x_2 \in 2\#5) \;=\; (\{x_1 = 1\}, \ldots, \{x_1 = 10\}).$$

Die Distribuierungsstrategie $\Delta_2$ teilt einen Bereich in zwei Hälften:

$$\Delta_2(x_1 = 2 \wedge x_2 \in 1\#4) \;=\; (\{x_2 \in 1\#2\},\ \{x_2 \in 3\#4\}).$$

Eine dritte Distribuierungsstrategie $\Delta_3$ wählt diejenige Variable aus, die einen kleinsten Bereich hat, und weist dieser Variablen den kleinsten Wert aus ihrem Bereich zu oder entfernt diesen Wert. Eine Variable mit einem kleinen Bereich kommt typischerweise in vielen Constraints vor und ihre Determinierung kann zu viel Propagierung führen (mehr zu dieser sogenannten first-fail Strategie im nächsten Kapitel).

$$\Delta_3(x_1 \in 1\#5 \wedge x_2 \in 2\#4) \;=\; (\{x_2 = 2\},\ \{x_2 \neq 2\}).$$

An der Distribuierungsregel $\Delta_3$ wird besonders deutlich, daß die Anwendungsreihenfolge von Distribuierungsregeln und anderen Regeln wichtig ist. Durch die Anwendung von Propagierungsregeln können Bereichsconstraints weiter verstärkt werden, was wertvolle Information für die Distribuierungsstrategie liefert. Erst wenn keine Propagierung mehr erfolgen kann, wird distribuiert, wodurch maximale Information zur Vefügung steht.

Die eingeführte Distribuierungsregel und die Definition von Distribuierungsstrategien lassen auch nichtterminierende Berechnungen zu. In der Praxis sind wir nur an Strategien interessiert, die nur eine endliche Zahl von Distribuierungsschritten zulassen. Für alle in dieser Arbeit vorgestellten Distribuierungsstrategien gilt diese Annahme. Eine Strategie erfüllt diese Annahme, wenn in einem Distribuierungsschritt von $(d,P)$ mit $(P_1,\ldots,P_m)$ jedes $P_i$ mindestens einen Projektor $p$ enthält, der bei Anwendung der Propagierungsregel zu einem echt stärkeren Bereichsconstraint als $d$ führt.

### Suchbäume

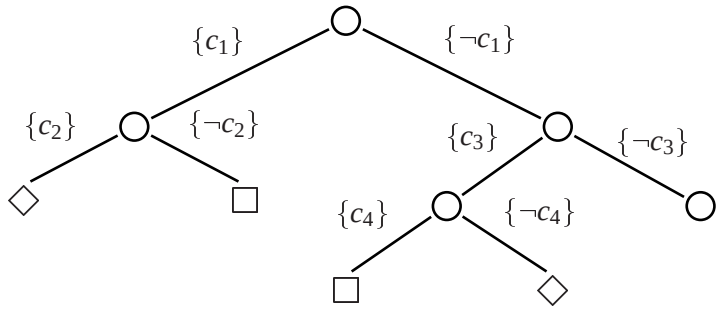
Berechnung auf Konfigurationen kann auch durch einen Baum veranschaulicht werden. Dabei repräsentiert ein Knoten eine Konfiguration und alle durch Anwendung von Normalformregeln daraus hervorgehenden Konfigurationen. Wird eine stabile Konfiguration $K=(d,P)$ mit $(P_1,\ldots,P_m)$ distribuiert, so bilden die Knoten, die $K_1=(d,P\cup P_1)$ bis $K_n=(d,P\cup P_m)$ enthalten, von links nach rechts die Nachfolger des $K$ enthaltenen Knotens im Baum (wir nehmen an, daß die $K_i$ paarweise verschieden sind). Wir sagen dann auch, daß ein Knoten durch Distribuierung *expandiert* wird. Der so entstehende Baum heißt *Suchbaum*. Ist keine Expansion mehr möglich, so nennen wir den entstandenen Suchbaum *vollständig*.

Enthalte die Wurzel eines Suchbaumes die Konfiguration $K$. Die Blätter eines vollständigen Suchbaumes enthalten entweder fehlgeschlagene Konfigurationen (sogenannte *Fehlerknoten*) oder gelöste Konfigurationen (sogenannte *Lösungsknoten*). Gilt $\{K\} \rightarrow^* \{K_1,\ldots,K_n\}$ und kann nicht weiter reduziert werden, so ist jede Konfiguration $K_i$ in einem Lösungsknoten enthalten. Ein Knoten, der eine stabile Konfiguration enthält, heißt *Wahlpunkt* (engl. *choice point*).

Gilt $\{K\} \rightarrow^* \{K_1,\ldots,K_n\}$, so bilden die Knoten des Suchbaumes, die $K_1$ bis $K_n$ enthalten, und die Fehlerknoten die *Front* des Suchbaumes.

Ein Beispiel für einen Suchbaum ist in Abbildung 2.3 zu sehen. An den Kanten stehen jeweils die Constraints, die durch die Projektoren realisiert werden, mit denen distribuiert wird. Rauten repräsentieren Lösungsknoten, Quadrate Fehlerknoten und Kreise Wahlpunkte. Wir werden oft die Größe von Suchbäumen als Komplexitätsmaß verwenden, um die Qualität von Modellen und Distribuierungsstrategien bei der Lösungssuche bestimmen zu können (siehe z. B. Kapitel 3).

**Abbildung 2.3** Ein Suchbaum



Wie ein Suchbaum durchlaufen wird, hängt von der gewählten *Suchstrategie* ab. Die Suchstrategie ist unabhängig von der Distribuierungsstrategie. Wenn nicht anders gesagt, gehen wir von einer linksorientierten *Tiefensuche* aus. Das heißt, es wird jeweils der Wahlpunkt durch Distribuierung expandiert, der am weitesten links in der Front des Suchbaumes steht (siehe auch Abschnitt 7.1.4). Für die Berechnung auf Konfigurationen entspricht dies der Tatsache, daß wir von Konfigurationsmengen zu Konfigurationslisten übergehen und Regeln nur für die am weitesten vorne stehende Konfiguration in dieser Liste anwenden, die nicht gelöst ist. Bei Tiefensuche ist der vollständige Suchbaum eindeutig durch die Konfiguration in der Wurzel und eine Distribuierungsstrategie bestimmt. Verwendet man eine Suchstrategie wie Branch-and-Bound, bei der während der Exploration des Suchbaums durch die Suchstrategie Projektoren in Konfigurationen hinzugefügt werden, gilt dies nicht mehr (siehe auch Abschnitt 3.2 und 7.1.4).

Die Berechnung von Lösungen soll an einem Beispiel verdeutlicht werden [SSW98]. Dazu sei die Spezifikation

$$(X \in 1\#8 \ \wedge \ Y \in 1\#10 \ \wedge \ Z \in 1\#10, \ \ \{X \neq 7, \ Z \neq 2, \ X - Z = 3 \cdot Y\})$$

gegeben. Die realisierende Konfiguration enthält den Bereichsconstraint der Spezifikation, vollständige Projektoren für die Constraints $X \neq 7$ und $Z \neq 2$, sowie einen Projektor mit dem gleichen Propagierungsverhalten wie die zwei vollständigen Projektoren für $X - Z \leq 3 \cdot Y$ und $X - Z \geq 3 \cdot Y$ zusammen (für Details siehe Abschnitt 4.1). Im Beispiel betrachten wir immer nur den sich verändernden Bereichsconstraint.

Die Projektoren für $X \neq 7$ und $Z \neq 2$ sind nach ihrer Anwendung sofort subsumiert. Es gilt dann

$$X \in [1\#6 \ \ 8] \ \wedge \ Y \in 1\#10 \ \wedge \ Z \in [1 \ \ 3\#10].$$

Die Projektoren für $X - Z = 3 \cdot Y$ können jetzt die Bereiche weiter zu

$$X \in [4\#6 \ \ 8] \ \wedge \ Y \in 1\#2 \ \wedge \ Z \in [1 \ \ 3\#5]$$

einschränken. Jetzt ist eine stabile Konfiguration erreicht und wir distribuieren mit $(\{X = 4\}, \ \{X \neq 4\})$. Abbildung 2.4 zeigt den insgesamt resultierenden Suchbaum. Dabei schreiben wir $\perp$ in einen Knoten, falls der Knoten eine fehlgeschlagene Konfiguration enthält. Ansonsten wird der Bereichsconstraint der im Knoten enthaltenen gelösten oder stabile Konfiguration gezeigt. Durch die Distribuierung mit $\{X = 4\}$ wird durch Propagierung eine gelöste Konfiguration erreicht:

$$X = 4 \ \wedge \ Y = 1 \ \wedge \ Z = 1.$$

Aus der Distribuierung mit $\{X \neq 4\}$ erhalten wir schließlich eine stabile Konfiguration mit dem Bereichsconstraint

$$X \in [5\#6 \ \ 8] \ \wedge \ Y \in 1\#2 \ \wedge \ Z \in [1 \ \ 3\#5].$$

Distribuieren wir mit $\{X = 5\}$, so erhalten wir eine fehlgeschlagene Konfiguration, da $\{X = 5\}$ mit $X - Z = 3 \cdot Y$ inkonsistent ist. Distribuieren wir mit $X \neq 5$, erhalten wir

$$X \in [6 \ \ 8] \ \wedge \ Y \in 1\#2 \ \wedge \ Z \in [1 \ \ 3\#5].$$

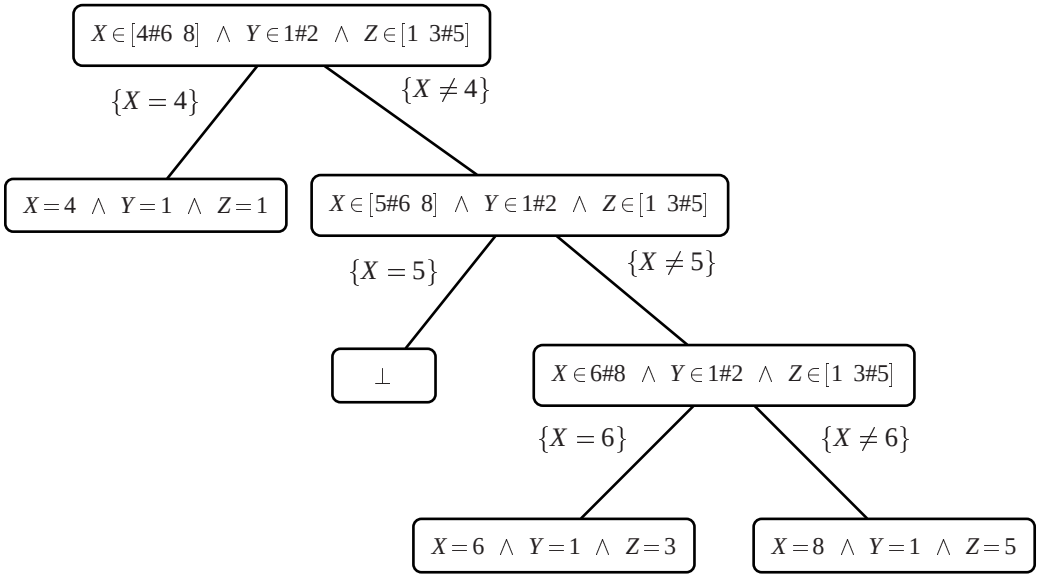
Jetzt distribuieren wir mit $\{X = 6\}$. Durch Propagierung erreichen wir eine gelöste Konfiguration mit den Bereichen

$$X = 6 \ \wedge \ Y = 1 \ \wedge \ Z = 3.$$

Letztendlich führt auch Distribuierung mit $\{X \neq 6\}$ zu einer gelösten Konfiguration:

$$X = 8 \ \wedge \ Y = 1 \ \wedge \ Z = 5.$$

**Abbildung 2.4** Ein Suchbaum mit 3 Wahlpunkten, 1 Fehlerknoten und 3 Lösungsknoten




### Zustandsabhängige Distribuierungsstrategien

Ein Anwendungsgebiet, indem die bisher vorgestellten Distribuierungsstrategien nicht ausreichen, ist Scheduling (Ablaufplanung). Bei den bisher vorgestellten Strategien zeigt die Determiniertheit einer Variablen an, daß ein Distribuierungsschritt für diese Variable nicht mehr nötig ist. Bei Scheduling werden jedoch nicht unbedingt in einem Distribuierungsschritt Variablen determiniert, sondern es werden Ordnungsentscheidungen getroffen, indem zum Beispiel mit Projektoren für Constraints wie $start(x) + dur(x) \leq start(y)$ distribuiert wird. Hierbei bezeichnen $x$ und $y$ zwei Aufgaben, $start(x)$ und $start(y)$ die entsprechenden Startzeiten und $dur(x)$ die Dauer von $x$. Somit bedeutet der Constraint $start(x) + dur(x) \leq start(y)$, daß $x$ vor $y$ plaziert wird. Die bloße Betrachtung des aktuellen Bereichsconstraints reicht nicht aus, um bereits getroffene Entscheidungen bestimmen zu können. Da wir aber nur an Distribuierungsstrategien interessiert sind, die eine endliche Zahl von Distribuierungsschritten zulassen, müssen wir es einer Distribuierungsstrategie ermöglichen, getroffene Ordnungsentscheidungen (effizient) zu erkennen.

Hierfür führen wir *zustandsabhängige Distribuierungsstrategien* ein. Dazu wird zu jeder Konfiguration ein Zustand $\sigma$ assoziiert, der nur von der Distribuierungsregel berücksichtigt wird. In diesem Zustand wird vermerkt, welche Distribuierungsschritte bis zu dieser Konfiguration schon ausgeführt worden sind. Die Distribuierungsstrategie wird dann den Zustand geeignet interpretieren, um zum Beispiel bereits getroffene Ordnungsentscheidungen zu berücksichtigen. Zustandsabhängige Distribuierungsstrategien umfassen insbesondere die Strategien nach Definition 10. Am Anfang einer Berechnung ist der Zustand für eine Konfiguration, die eine Spezifikation realisiert, leer. Sei nun $\mathcal{S}$ die Menge aller Zustände.

**Definition 11** *Eine* $m$-stellige zustandsabhängige Distribuierungsstrategie $(\Delta, \sigma_0)$ *besteht aus ei-*

*ner Funktion*

$$\Delta : \mathcal{D} \times \mathcal{S} \rightarrow (\mathcal{P}, \mathcal{S})^m$$

*für* $m \geq 0$*, und einem Anfangszustand* $\sigma_0$*. Es gilt*

$$\{(d, P, \sigma), \ldots\} \rightarrow \{(d, P \cup P_1, \sigma_1), \ldots, (d, P \cup P_m, \sigma_m), \ldots\}$$

*genau dann, wenn* $\Delta(d, \sigma) = ((P_1, \sigma_1), \ldots, (P_m, \sigma_m))$ *gilt.*

Beachte, daß bei der Anwendung der Distribuierungsregel für zustandsabhängige Distribuierungsstrategien die Anzahl der Konfigurationen in einer Konfigurationsmenge auch gleich bleiben kann (im Fall $m = 1$). Da in diesem Fall dann $d \wedge c(P) \models c(P_1)$ gilt, werden also redundante Projektoren zu einer Konfigurationen hinzugefügt. Diese Projektoren können dann die Propagierung in der "distribuierten" Konfiguration verstärken. Beachte, daß die Regeln auf Konfigurationen nur Bereichsconstraints aber nicht die Projektorenmenge verändern.

Im Fall $m = 0$ wird eine Konfiguration $(d, P, \sigma)$ aus der Konfigurationsmenge entfernt, also $\{(d, P, \sigma), K_1, \ldots\} \rightarrow \{K_1, \ldots\}$. In einem solchen Fall gilt nämlich $d \wedge c(P) \models \bot$. Da die Konfiguration $(d, P, \sigma)$ (genauer $d \wedge c(P)$) somit keine Lösung besitzt, kann diese Konfiguration auch entfernt werden. Die beiden Fälle $m = 0$ und $m = 1$ werden für Schedulinganwendungen verwendet (siehe Kapitel 6).

## 2.7 Bemerkungen und verwandte Arbeiten

Grundlegende Ideen für Constraintpropagierung stammen aus dem Gebiet der Constraintnetze [Mac77, Mon74, Wal75, Tsa93]. Hierbei soll für eine Menge von Constraints herausgefunden werden, ob sie erfüllbar ist (engl. constraint satisfaction problems; traditionell wird dieses Feld der Künstlichen Intelligenz zugeordnet). Es wurde eine Reihe von Algorithmen vorgeschlagen, um Normalformen einer Menge von Constraints bzgl. einer Menge von Variablenbereichen zu berechnen (von AC-3 [Mac77] über AC-5 [VDT92] bis zu mindestens AC-7 [BFR95]). Für Constraints wird in diesen Arbeiten meistens die Projektion berechnet. Doch gibt es auch Ansätze, die nur eine Approximation einer Projektion berechnen (siehe [Nad88] für einen Überblick). Für Constraintnetze wird nicht zwischen einem Constraint als Relation und einer Propagierungsfunktion zur Constraintpropagierung unterschieden. Constraints werden in der Regel explizit durch eine Menge von Wertekombinationen repräsentiert (extensionale Repräsentation), weshalb realistische Probleme aus der Praxis aus Komplexitätsgründen nicht lösbar sind. Algorithmus 2.1 ist eine Verallgemeinerung des AC-3 Algorithmus auf Constraints mit beliebig vielen Argumenten.

In [GH88] wird ein Constraint mit einer Funktion auf Bereichen von Variablen assoziiert, einem sogenannten Filteroperator. Für diese Filteroperatoren werden Korrektheit, Extensionalität und Monotonie gefordert. Constraintpropagierung wird mit diesen Operatoren beschrieben. Es wird jedoch nicht darauf eingegangen, daß die Operatoren in unserer Sprechweise auch unvollständig sein können (und für effiziente Implementierungen auch sein sollten).

Mit der logischen Constraintsprache CHIP [DVS+88] werden die für Constraintnetze entwickelten Konzepte aufgegriffen und modifiziert, um effizient kombinatorische Probleme lösen zu können. Insbesondere wird meist auf die Berechnung von Projektionen zugunsten von schwächerer Propagierung [Van89] verzichtet. Wegweisend ist in CHIP aber das Ausnutzen komplexer

algorithmischer Techniken für spezifische Constraints wie für Scheduling oder Plazierungsprobleme [DSV90, AB93]. Zudem wird eine Unterscheidung in primitive Constraints (bei uns sind dies die Bereichsconstraints) und nicht-primitive Constraints (bei uns ist dies die Menge der Constraints in einer Spezifikation oder die Projektoren in einer Konfiguration) vorgenommen [VD91a], die in praktisch alle späteren System übernommen wird. Für Propagierung und Distribuierung gibt es in CHIP nur ein (von Prolog) sprachabhängiges Berechnungsmodell [Van89].

Für die Ansätze, die sogenannte Indexicals zur Propagierung verwenden [CD96, Car95], wird ein Berechnungsmodell angegeben, das auf nebenläufiger Constraintprogrammierung [Sar93] beruht. Das Berechnungsmodell ist aber direkt auf Indexicals zugeschnitten, während unser Modell unabhängig von einer konkreten Formulierung von Projektoren ist. Distribuierung wird aus dem Modell ausgespart und erscheint lediglich als Hilfsmittel der die Indexicals beherbergenden Programmiersprache.

Die Programmiersprache `cc(FD)` [VSD95, Van94] kommt unserem Ansatz näher, da keine Indexicals verwendet werden und die Sprache auch auf nebenläufiger Constraintprogrammierung beruht. Die Propagierung für Constraints beruht auf Intervall- und Bereichspropagierung (siehe auch Seite 51) und ist damit nicht so allgemein wie unser Modell. Auch hier erscheint Distribuierung nicht in dem Modell.

In [Ben96, BG96] werden mit Constraints sogenannte narrowing-Operatoren assoziiert, die die Propagierung realisieren. Solche Operatoren und ihre Eigenschaften kommen den hier verwendeten Projektoren sehr nahe. Die Berechnung einer Normalform geschieht durch einen sogenannten Filteralgorithmus. Narrowing-Operatoren, die reduktionsvollständig sind (also eine Projektion realisieren), heißen in [Ben96] optimale Operatoren. Ein Operator heißt vollständig in [Ben96], wenn in unserer Schreibweise aus $d \models \neg c$ auch $N_p(d) = \bot$ folgt. In diesen Arbeiten werden jedoch keine Tests auf Subsumtion oder Dissubsumtion verwendet, und es wird nicht auf Distribuierung eingegangen.

Für das Gebiet der Constraintnetze besteht Distribuierung aus einer Variablenselektion und anschließender Selektion eines Werts aus dem Variablenbereich, mit dem die Variable dann belegt wird [Kum92, Tsa93]. Die Verwendung von beliebigen Constraints bzw. Projektoren zur Distribuierung ist inhärent in (logischer) Constraintprogrammierung vorhanden, da ein Prädikat durch mehrere Klauseln definiert werden kann, so daß eine Fallunterscheidung ermöglicht wird (siehe z. B. [Van89]).

Die Idee, Propagierung und Distribuierung so zu verknüpfen, daß erst distribuiert wird, nachdem keine weitere Propagierung mehr möglich ist, geht auf Konzepte für Constraintnetze zurück ([Wal75]; siehe [Kum92] für eine Übersicht).

# Kapitel 3

# Beispiele

In diesem Kapitel erläutern wir anhand einiger Beispiele das Zusammenspiel von Propagierung und Distribuierung zur Lösung von kombinatorischen Problemen. Um zu zeigen, wie die Auswahl von Projektoren und Distribuierungsstrategien die Effizienz der Lösungssuche beeinflußt, werden wir für jedes Problem mehrere Modelle und Distribuierungsstrategien miteinander vergleichen. Bei mehreren Beispielen werden wir die Vorteile von redundanten Constraints und dem Ausschluß von Symmetrien untersuchen. Während die Probleme in diesem Kapitel unabhängig von einer konkreten Implementierung betrachtet werden, sind in Kapitel 7 einige der Probleme in Oz implementiert.

Im ersten Teil dieses Kapitels betrachten wir eine Reihe von Puzzles. Im zweiten Teil beschäftigen wir uns mit Schedulingproblemen.

## 3.1 Puzzles

### 3.1.1 Zahlenpuzzles

**Send More Money** Zuerst greifen wir das Zahlenpuzzle aus Abschnitt 2.2 wieder auf. Es sind verschiedene Ziffern für die Buchstaben $S$, $E$, $N$, $D$, $M$, $O$, $R$, $Y$ zu finden, so daß die Gleichung $SEND + MORE = MONEY$ gilt sowie $S$ und $M$ von Null verschieden sind (keine führenden Nullen). Die einzige Lösung dieses Problems ist $9567 + 1085 = 10652$. Der Suchraum besteht aus $10^8$ verschiedenen Variablenbelegungen.

Für ein Modell dieses Problems führen wir für jeden Buchstaben eine Variable ein, die für die mit dem Buchstaben assoziierte Ziffer steht, und übernehmen die Constraints der Spezifikation aus Abschnitt 2.2, so daß für jede Variable ein Bereich 0#9 vorliegt. Wir wählen vollständige Projektoren für die Constraints $S \neq 0$ und $M \neq 0$. Der Constraint, daß alle Ziffern verschieden sein müssen, wird durch einen Projektor realisiert, der Werte von determinierten Variablen aus den Bereichen der übrigen Variablen entfernt (siehe auch Seite 21). Die Gleichung des Problems wird durch den Constraint

$$1000 \cdot S + 100 \cdot E + 10 \cdot N + D$$

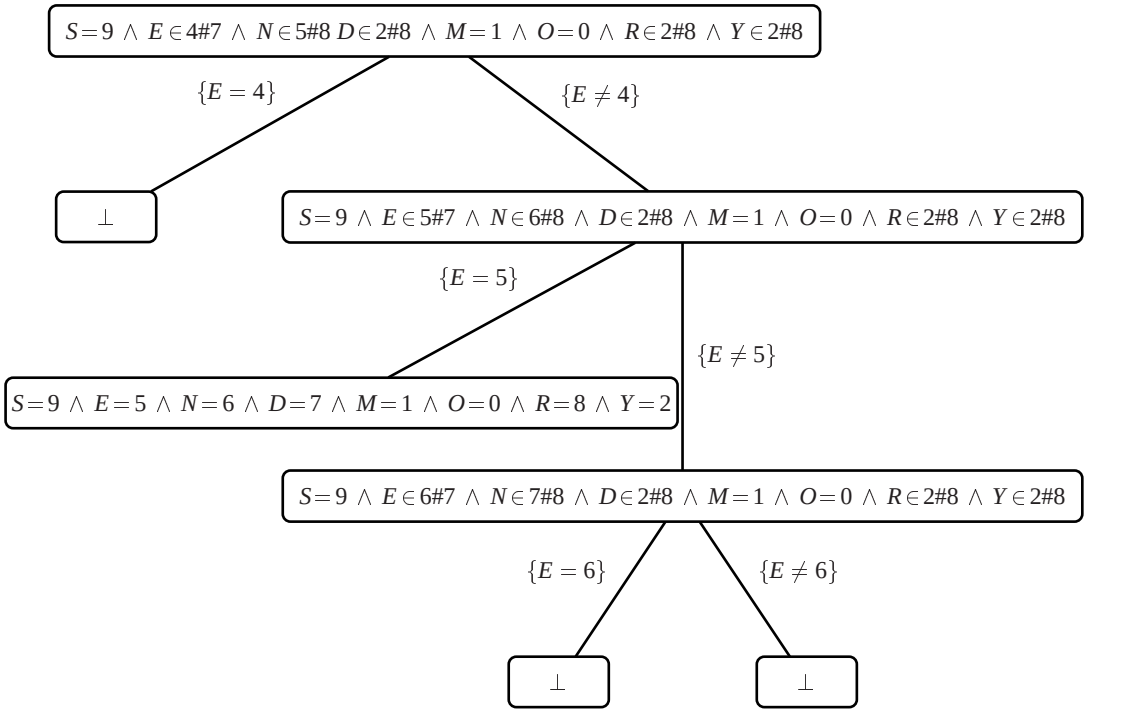
$$\begin{array}{rrl} + & & 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = & 10000 \cdot M + & 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{array}$$

beschrieben. Ist $s = t$ dieser Constraint, so wird er durch einen Projektor realisiert, der das gleiche Propagierungsverhalten hat wie die zwei vollständigen Projektoren für $s \leq t$ und $s \geq t$ zusammen (der Projektor ist also unvollständig; siehe Seite 23).

Wir wählen eine Distribuierungsstrategie, die die am weitesten links stehende, noch nicht determinierte Variable $x$ in $S, E, N, D, M, O, R, Y$ auswählt und mit $(\{x = n\},\ \{x \neq n\})$ distribuiert ($n$ ist der kleinste Wert im Bereich von $x$).

Der durch Propagierung und Distribuierung entstehende Suchbaum ist in Abbildung 3.1 gezeigt. Beachte, daß $S$, $M$ und $O$ bereits durch Propagierung determiniert sind. Durch Distribuierung mit

**Abbildung 3.1** Suchbaum für das Problem $SEND + MORE = MONEY$




$\{E \neq 4\}$ und dann $\{E = 5\}$ erhalten wir eine Lösung. Insgesamt wurde der Suchraum von $10^8$ möglichen Kombinationen auf einen Suchbaum mit 7 Knoten reduziert.

Realisieren wir die Gleichung durch einen vollständigen Projektor, so wird die Lösung in einem Suchbaum mit einem Wahlknoten und einem Fehlerknoten gefunden. Die Verwendung des vollständigen Projektors läßt die Laufzeit in einer konkreten Implementierung wie DFKI Oz [Pro97] jedoch um fast zwei Größenordnungen anwachsen. Wird auch ein vollständiger Projektor für den Constraint verwendet, der besagt, daß alle Variablenwerte paarweise verschieden sein müssen, wird die Lösung allein durch Propagierung gefunden.

**Bruchrechnung** In einem zweiten Zahlenpuzzle [SSW98] müssen verschiedene Ziffern für die Buchstaben *A* bis *I* gefunden werden, so daß die folgende Gleichung gilt.

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

Wie im vorangehenden Beispiel führen wir für jeden Buchstaben eine Variable ein (mit dem Bereich 0#9), die für die mit dem Buchstaben assoziierte Ziffer steht. Nun führen wir für $BC$ etc. Hilfsvariablen $H_{BC}$ etc. ein. Für die Hilfsvariablen gelten dann Constraints wie $10 \cdot B + C = H_{BC}$ etc. und die Bereiche dieser Hilfsvariablen sind 0#99. Nun kann die Gleichung durch den Constraint

$$A \cdot H_{EF} \cdot H_{HI} + D \cdot H_{BC} \cdot H_{HI} + G \cdot H_{BC} \cdot H_{EF} = H_{BC} \cdot H_{EF} \cdot H_{HI}$$

spezifiziert werden.

Wir werden nun zwei Möglichkeiten vorstellen, um den Suchraum weiter zu reduzieren. Da wir aber nur an prinzipiell verschiedenen Lösungen interessiert sind und die Brüche sich nicht weiter unterscheiden, sollten wir symmetrische Lösungen durch Ordnungsconstraints

$$\frac{A}{BC} \geq \frac{D}{EF} \geq \frac{G}{HI}$$

ausschließen. Aus den Ordnungsconstraints können wir jetzt die folgenden *redundanten* Constraints ableiten.

$$3 \cdot \frac{A}{BC} \geq 1 \quad \text{und} \quad 3 \cdot \frac{G}{HI} \leq 1$$

Diese Constraints werden zwar von den Ordnungsconstraints subsumiert. Die zugehörigen Projektoren können den Suchraum aber stärker einschränken als es die Projektoren der Ordnungsconstraints alleine vermögen. Diese Constraints lassen sich auch wieder mit den Hilfsvariablen geeignet umformulieren.

Nach der Formulierung der Constraints legen wir nun die Projektoren fest. Dabei realisieren wir einen Constraint $s = t$ wieder durch einen Projektor mit dem Propagierungsverhalten der zwei vollständigen Projektoren für $s \leq t$ und $s \geq t$. Die Ungleichungen des Problems werden durch vollständige Projektoren realisiert. Zusätzlich haben wir einen Projektor, der für eine determinierte Variable deren Wert aus den Bereichen der übrigen Variablen entfernt. Wir wählen die gleiche Distribuierungsstrategie wie im vorherigen Beispiel (für die Liste von Variablen $A, B, C, D, E, F, G, H, I$). Durch die Determinierung dieser Variablen werden die Hilfsvariablen auomatisch determiniert.

Die erste Lösung wird nach 743 Wahlpunkten gefunden. Läßt man die redundanten Constraints weg, so sind 2 201 Wahlpunkte erforderlich. Berücksichtigt man zusätzlich auch die Ordnungsconstraints nicht, so benötigt man 2 663 Wahlpunkte.

Deutlicher wird der Einfluß der Ordnungsconstraints, wenn man man nach allen Lösungen sucht (wobei es jedoch nur eine gibt). Die vorgestellte Realisierung benötigt 1 131 Wahlpunkte. Ohne die redundanten Constraints erhöht sich diese Zahl auf 2 912 und läßt man auch noch die Ordnungsconstraints weg, so benötigt man 9 831 Wahlpunkte.

Realisieren wir den Constraint, daß alle Variablen verschiedene Werte annehmen müssen, durch einen vollständigen Projektor, so sind immer noch 1 128 Wahlpunkte zum Finden aller Lösungn

notwendig. Realisieren wir noch die Gleichung $10 \cdot B + C = H_{BC}$ etc. vollständig[1], so werden noch 1 126 Wahlpunkte benötigt (bei einem Anstieg der Laufzeit um etwa den Faktor Drei). Für dieses Beispiel bringen vollständige Projektoren keinen wirklichen Gewinn, da ihre zusätzliche Information von den anderen Projektoren nicht genutzt werden kann.

### 3.1.2 Das *n*-Damen-Problem

Das sogenannte *n*-Damen-Problem besteht darin, $n$ Damen auf einem $n \times n$ Schachbrett so zu plazieren, daß sie sich gegenseitig nicht schlagen können (siehe z. B. [Van89] für Constraintprogrammierung und [FS86] für einen Algorithmus, der eine Lösung ohne Suche konstruiert). Eine Lösung für $n = 5$ ist in Abbildung 3.2 gezeigt.

**Abbildung 3.2** Eine Lösung des 5-Damen-Problems

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | | | $x_3$ | | |
| 2 | $x_1$ | | | | |
| 3 | | | | $x_4$ | |
| 4 | | $x_2$ | | | |
| 5 | | | | | $x_5$ |

Es ist naheliegend, die Position einer Dame durch zwei Variablen zu modellieren, die deren Koordinaten auf dem Brett angeben. Jedoch sind hier Symmetrien enthalten. Wird zum Beispiel in einer Lösung die erste Dame auf $(1,2)$ und die zweite auf $(2,4)$ plaziert, so unterscheidet sich diese Lösung von einer Lösung mit der Plazierung $(2,4)$ und $(1,2)$ nur durch die Reihenfolge der Damen.

Um diese Symmetrien zu entfernen, führen wir im Modell für jede Spalte eine Variable $x_i$ ein, die die Zeilenposition der zugehörigen Dame angibt. Dies garantiert auch, daß niemals zwei Damen in der gleichen Spalte plaziert werden. Für jede Variable existiert ein Bereichsconstraint $x_i \in 1\#n$.

Daß je zwei Damen sich nicht gegenseitig schlagen dürfen, läßt sich durch Ungleichconstraints über zwei Variablen ausdrücken. Die Ungleichconstraints können durch Projektoren realisiert werden, die den Wert einer determinierten Variablen aus dem Bereich der anderen entfernen.

(1) $x_i \neq x_j$ für gleiche Zeilen
(2) $x_i - i \neq x_j - j$ für Diagonalen von links oben nach rechts unten
(3) $x_i + i \neq x_j + j$ für Diagonalen von links unten nach rechts oben

mit $i$, $j$, so daß $1 \leq i < j \leq n$ gilt.

Wir wählen als Distribuierungsstrategie die naive Strategie aus dem vorangehenden Beispiel (be-

[1] Für nichtlineare Gleichungen (also Gleichungen, in denen Produkte von Variablen vorkommen) ist in Oz kein vollständiger Projektor verfügbar.

ginnend bei $x_1$). Um die erste Lösung des Problems zu finden, ergibt sich für das 16-Damen-Problem ein Suchbaum mit 1 842 Wahlpunkten.

Die naive Distribuierungsstrategie geht nicht auf die dynamische Entwicklung der Bereiche der Variablen ein. Alternativ kann die aktuelle Größe eines Variablenbereichs zur Grundlage der Variablenselektion gemacht werden. Hat eine Variable einen kleinen Bereich, so kann dies ein Indiz dafür sein, daß die Variable in sehr vielen Projektoren vorkommt und ihre Determinierung viel Propagierung zur Folge haben kann. Gibt es in dem gerade betrachteten Teilbaum eine Lösung sollte sie bei dieser Wahl schneller gefunden werden, als bei der Wahl einer anderen Variablen. Gibt es in dem untersuchten Teilbaum jedoch keine Lösung, so ist es günstig, dies möglichst früh zu entdecken. Unter der Annahme, daß die möglichen Werte in Bereichen mit gleicher Wahrscheinlichkeit Teil einer Lösung sind, sollte es für eine Variable mit wenigen Werten wahrscheinlicher sein, daß ihre Determinierung das Fehlen von Lösungen im Teilbaum aufzeigt. Diese Strategie nennt man deshalb *first-fail* Strategie (siehe z.B. [BR75] oder [HE80]). Auch hier wird zuerst der kleinste Wert in dem Bereich zur Distribuierung verwendet.

Mit der first-fail Strategie ergibt sich ein Suchbaum mit nur 16 Wahlpunkten, um eine erste Lösung zu finden; eine Reduktion um mehr als zwei Größenordnungen.

Schaut man sich die Constraints für die Zeilen und Diagonalen noch einmal genauer an, so fällt auf, daß diese auch anders formuliert werden können:

| | | |
|---|---|---|
| (1) | $x_1, \ldots, x_n$ | paarweise verschieden |
| (2) | $x_1 - 1, \ldots, x_n - n$ | paarweise verschieden |
| (3) | $x_1 + 1, \ldots, x_n + n$ | paarweise verschieden |

Bei dieser Formulierung werden $(3 \cdot n \cdot (n-1))/2$ Constraints durch drei Constraints ersetzt. Realisiert man jeden dieser Constraints durch einen einzelnen Projektor, der das Propagierungsverhalten hat, wie es in den vorangehenden Abschnitten geschildert wurde (entsprechend erweitert für den Fall der Addition/Subtraktion von Konstanten), schlägt sich dies in einer konkreten Implementierung wie DFKI Oz in einer Halbierung der Laufzeit nieder. Verwendet man für den Constraint, daß $x_1$ bis $x_n$ verschiedene Werte annehmen müssen, einen vollständigen Projektor, so werden bei der naiven Distribuierungsstrategie aus Abschnitt 3.1.1 nur noch 1 448 Wahlpunkte benötigt (jedoch bei dem 1.6-fachen der vorherigen Laufzeit). Bei der first-fail Strategie benötigt man immer noch 16 Wahlpunkte (obwohl diese Strategie von der stärkeren Information der vollständigen Projektoren profitieren könnte). Wir sehen hier wieder, daß unvollständige Projektoren sehr wohl gute Resultate aufgrund ihrer Effizienz und der oft ausreichend guten Propagierung liefern können.

### 3.1.3 Magische Folgen

In diesem Beispiel [Van89] wird noch einmal der Nutzen von redundanten Constraints verdeutlicht. Eine Folge

$$x_0, x_1, \ldots, x_{n-1}$$

von natürlichen Zahlen heißt magisch, wenn für jedes $i \in \{0, \ldots, n-1\}$ gilt:

- $x_i$ ist eine natürliche Zahl zwischen 0 und $n-1$ und

- die Zahl $i$ kommt genau $x_i$-mal in der Folge vor.

Wir modellieren das Problem durch $n$ Variablen mit dem Bereich $0\#n$. Der Constraint über das Vorkommen eines $x_i$ in der Folge wird durch einen vollständigen Projektor realisiert. Wählen wir diejenige Distribuierungsstrategie, die die Variablen von $x_0$ beginnend mit dem kleinstmöglichen Wert determiniert, so benötigen wir für $n = 50$ genau 184 Wahlpunkte, um die erste Lösung zu finden.

Nun kann man sich leicht überlegen, daß für magische Folgen die Gleichungen

$$x_0 + \ldots + x_{n-1} = n$$
$$0 \cdot x_0 + 1 \cdot x_1 + \ldots (n-1) \cdot x_{n-1} = n$$

gelten. Fügt man dem bisherigen Modell noch einen unvollständigen Projektor für den ersten redundanten Constraint hinzu, so benötigt man nur noch 116 Wahlpunkte. Fügt man nur den unvollständigen Projektor für den zweiten redundanten Constraint hinzu, benötigt man 94 Wahlpunkte. Mit beiden Projektoren zusammen benötigt man nur 70 Wahlpunkte. Eine Realisierung der redundanten Constraints durch vollständige Projektoren kommt wegen der großen Zahl von Variablen aus Komplexitätsgründen nicht in Frage.

Verwendet man first-fail als Distribuierungsstrategie, so fällt der Vorteil redundanter Constraints noch stärker auf. Für die obigen Fälle benötigen wir 1 368, 96, 27 und nochmals 27 Wahlpunkte, um die erste Lösung zu finden.

## 3.2 Scheduling

In diesem Abschnitt werden anwendungsorientierte Probleme aus dem Gebiet des *Scheduling* (engl. für *Ablaufplanung*) betrachtet. Vereinfacht bedeutet Scheduling das Erstellen eines Zeitplans für Aufgaben, die um eine gegebene Menge von Ressourcen konkurrieren. Wir nehmen an, daß die Ausführung einer Aufgabe nicht unterbrochen werden kann. Auf Schedulingprobleme werden wir noch in den Kapiteln 5 und 6 ausführlicher eingehen.

### 3.2.1 Hausbau

Das erste Problem besteht darin, ein Haus zu bauen (wobei wir einige Vereinfachungen vornehmen). Die einzelnen Aufgaben, deren Ausführungszeit (in Tagen) und die ausführende Firma sind in Tabelle 3.1 gegeben.

Hierbei bedeutet zum Beispiel der Eintrag, daß *A Vorgänger* von *B* ist, daß die Aufgabe *A* beendet sein muß, bevor mit der Ausführung von *B* begonnen werden kann. Für die Modellierung sollen die ausführenden Firmen zuerst nicht berücksichtigt werden.

Jede Aufgabe wird durch eine Variable modelliert, die ihren Beginn (ihre Startzeit) charakterisiert. Für diese Variablen werden wir den gleichen Buchstaben verwenden wie für die entsprechende Aufgabe. Als frühesten Starttermin für eine Aufgabe nehmen wir 0 an. Eine triviale obere Schranke für die notwendige Gesamtzeit ergibt sich durch Aufsummieren aller Ausführungszeiten (hier 29). Somit resultiert für jede Variable der Bereich $0\#29$. Aus der Vorgängerrelation

**Tabelle 3.1** Hausbau

| Aufgabe | Inhalt | Ausführungszeit | Vorgänger | Firma |
|---|---|---|---|---|
| $A$ | Mauern errichten | 7 | - | Karl |
| $B$ | Dachstuhl bauen | 3 | $A$ | Otto |
| $C$ | Dach decken | 1 | $B$ | Otto |
| $D$ | Installationen | 8 | $A$ | Karl |
| $E$ | Außenputz | 2 | $C,D$ | Karl |
| $F$ | Fenster | 1 | $C,D$ | Otto |
| $G$ | Garten anlegen | 1 | $C,D$ | Otto |
| $H$ | Decken einziehen | 3 | $A$ | Karl |
| $I$ | Malerarbeiten | 2 | $F,H$ | Fritz |
| $J$ | Umziehen | 1 | $I$ | Fritz |

ergeben sich eine Menge von sogenannten *Präzedenzconstraints*:

$$\left\{\begin{array}{lllll} A+7\leq B, & B+3\leq C, & A+7\leq D, & C+1\leq E, & D+8\leq E, \\ C+1\leq F, & D+8\leq F, & C+1\leq G, & D+8\leq G, & A+7\leq H, \\ & F+1\leq I, & H+3\leq I, & I+2\leq J. & \end{array}\right\}$$

Hierbei bedeutet zum Beispiel $A+7\leq B$, daß $B$ frühestens 7 Tage nach dem Start von $A$ begonnen werden darf. Diese Constraints werden durch vollständige Projektoren realisiert. Für unser Beispiel $A+7\leq B$ ergibt sich z. B., daß die Anwendung des zugehörigen Projektors den Bereich von $B$ auf 7#29 und den von $A$ auf 0#22 reduziert.

Insgesamt ergibt sich durch Propagierung eine Konfiguration mit dem Bereichsconstraint

$$A\in 0\#11 \ \wedge\ B\in 7\#22 \ \wedge\ C\in 10\#25 \ \wedge\ D\in 7\#18 \ \wedge\ E\in 15\#29 \ \wedge\ F\in 15\#26$$
$$G\in 15\#29 \ \wedge\ H\in 7\#24 \ \wedge\ I\in 16\#27 \ \wedge\ J\in 18\#29.$$

Verwendet man die naive Distribuierungsstrategie aus Abschnitt 3.1.1, so erhält man die folgende Lösung (wir geben hier und im folgenden den Bereichsconstraint an, der die Lösung definiert; siehe Seite 28):

$$A=0 \ \wedge\ B=7 \ \wedge\ C=10 \ \wedge\ D=7 \ \wedge\ E=15 \ \wedge\ F=15 \ \wedge\ G=15 \ \wedge\ H=7 \ \wedge\ I=16 \ \wedge\ J=18.$$

Dies entspricht auch der Lösung mit dem frühesten Umzugstermin.

Nun sollen die ausführenden Firmen berücksichtigt werden. Wir nehmen an, daß jede Firma nur eine Aufgabe gleichzeitig erledigen kann. Mit anderen Worten, je zwei Aufgaben $A$ und $B$, die eine Firma ausführt, dürfen sich zeitlich in ihrer Ausführung nicht überlappen. Es muß also für alle Aufgaben einer Firma (man bezeichnet die Firma auch als *Ressource*) eine *Serialisierung* gefunden werden (für jedes Paar $(A,B)$ muß entschieden werden, ob $A$ vor $B$ fertig wird oder umgekehrt).

Wir beschreiben die Serialisierung einer Ressource durch Constraints. Hierzu nehmen wir an, daß für jedes Aufgabenpaar $(x,y)$, so daß $x$ und $y$ durch die gleiche Firma ausgeführt werden, ein Projektor den Constraint

$$x+dur(x)\leq y \ \vee\ y+dur(y)\leq x$$

realisiert. Dabei beschreibt $dur(x)$ bzw. $dur(y)$ die Ausführungszeit von $x$ bzw. $y$. Einen solchen Constraint nennt man auch *Kapazitätsconstraint* bzw. *disjunktiven Constraint*, da die Kapazität der Ressource nicht überschritten werden darf bzw., da entweder $A$ vor $B$ oder $B$ vor $A$ plaziert werden muß.

Wir nehmen das folgende Propagierungsverhalten des Projektors für den Kapazitätsconstraint an. Der Projektor reduziert erst dann Bereiche, wenn entweder $x + dur(x) \leq y$ oder $y + dur(y) \leq x$ vom aktuellen Bereichsconstraint dissubsumiert ist. Ist $x + dur(x) \leq y$ dissubsumiert, so propagiert der Projektor wie ein vollständiger Projektor für den Constraint $y + dur(y) \leq x$. Ist $y + dur(y) \leq x$ dissubsumiert, propagiert der Projektor wie ein vollständiger Projektor für $x + dur(x) \leq y$.

Bei Verwendung der first-fail Distribuierungsstrategie für die Startzeitvariablen ergibt sich die erste Lösung zu

$$A=0 \wedge B=7 \wedge C=10 \wedge D=7 \wedge E=18 \wedge F=15 \wedge G=16 \wedge H=15 \wedge I=18 \wedge J=20$$

mit einem Suchbaum mit 11 Wahlpunkten und keinem Fehlerknoten.

Nun ist man aber auch in diesem Fall an einer optimalen Lösung interessiert, in der der Umzugstermin so früh wie möglich ist. Eine naive Herangehensweise ist es, alle Lösungen zu berechnen und dann die optimale darunter herauszufinden (hier gibt es insgesamt 6 Lösungen; im folgenden Abschnitt wird aber gezeigt, daß dieses Vorgehen im allgemeinen nicht machbar ist). Ein sehr viel besseres Vorgehen nutzt die Bewertung einer bereits gefundenen Lösung, um den Suchraum frühzeitig einzuschränken. Eine solche Strategie ist die sogenannte *Branch-and-Bound* Suchstrategie. Sei dazu $\{(d_1, P_1), \ldots, (d_n, P_n)\}$ die aktuell berechnete Menge von Konfigurationen und sei $s$ der Zeitpunkt des Umzugs in der ersten gefundenen Lösung. Nun fügt man in unserem Beispiel den Projektor für $J < s$ zu jeder Konfiguration $(d_i, P_i)$ hinzu, die weder fehlgeschlagen noch erfolgreich ist. Dies stellt sicher, daß nur Lösungen gefunden werden, die besser als die mit $s$ bewertete sind. Für solche besseren Lösungen fügt man den entsprechenden Projektor des stärkeren Constraints über die verbesserte Umzugszeit hinzu (und so weiter). Auf diese Weise findet man die optimale Lösung (und beweist deren Optimalität), ohne in der Regel alle möglichen Lösungen zu berechnen. Hierbei heißt 'beweisen', daß in der zuletzt betrachteten Konfigurationsmenge keine Konfiguration mehr reduziert werden kann.

Bei Benutzung der Branch-and-Bound Suchstrategie finden wir sofort heraus, daß die zuerst gefundene Lösung gleichzeitig auch die optimale Lösung ist. Inklusive des Beweises der Optimalität erzeugt Branch-and-Bound einen Suchbaum mit 11 Wahlpunkten und 11 Fehlerknoten.

Anstatt first-fail als Distribuierungsstrategie zu verwenden, können wir auch die Serialisierung einer Ressource durch eine Distribuierungsstrategie erreichen, die Aufgaben auf einer Ressource ordnet. Dies resultiert in einer Distribuierungsstrategie, die für ein Paar von Aufgaben $(x, y)$, so daß $x$ und $y$ die gleiche Ressource (Firma) benötigen, zuerst mit dem Projektor $p_1$ für den Constraint $x + dur(x) \leq y$ und dann mit dem Projektor $p_2$ für den Constraint $y + dur(y) \leq x$ distribuiert. Ist $d$ der Bereichsconstraint und $P$ die Projektormenge einer zu distribuierenden Konfiguration, so gilt für den Distribuierungsschritt $d \wedge c(P) \models c(\{p_1\}) \vee c(\{p_2\})$, da die Konfiguration die Kapazitätsconstraints realisiert. Serialisiert man die Aufgaben nacheinander für die Firmen Karl, Otto und Fritz und distribuiert dann gemäß der naiven Strategie zur Determinierung der

Startzeitvariablen aus Abschnitt 3.1.1, so ergibt sich die Lösung[2]

$$A=0 \wedge B=7 \wedge C=10 \wedge D=7 \wedge E=15 \wedge F=15 \wedge G=16 \wedge H=17 \wedge I=20 \wedge J=22.$$

Der Suchbaum enthält 24 Wahlpunkte und keinen Fehlerknoten.

Beachte, daß wir an dieser Stelle zustandsabhängige Distribuierungsstrategien benötigen, die erkennen, welche Ordnungsentscheidungen bereits getroffen wurden (siehe Abschnitt 2.6). Sind alle Ordnungsentscheidungen getroffen, berücksichtigt die Distribuierungsstrategie nur noch Bereichsconstraints, um Variablen zu determinieren.

Bei Benutzung der Branch-and-Bound Suchstrategie ergibt sich die optimale Lösung zu

$$A=0 \wedge B=7 \wedge C=10 \wedge D=7 \wedge E=18 \wedge F=15 \wedge G=16 \wedge H=15 \wedge I=18 \wedge J=20$$

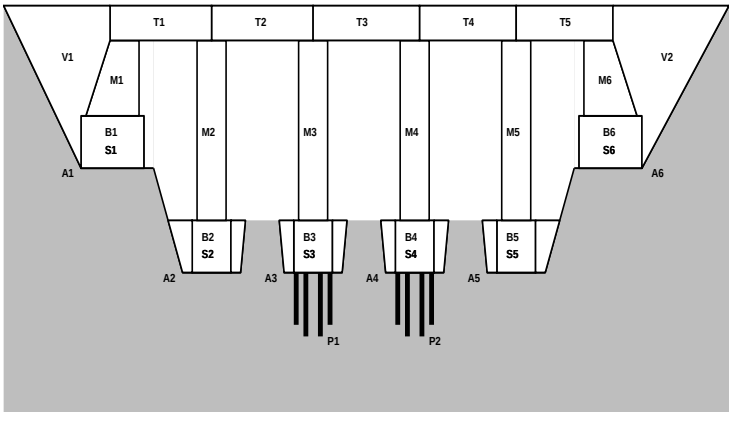
mit 35 Wahlpunkten, 34 Fehlerknoten und 2 Lösungsknoten (inkl. Optimalitätsbeweis).

### 3.2.2 Brückenbau

Das folgende Beispiel ist größer und realistischer als das vorangehende. Das Originalproblem stammt aus [Bar83]. In die Constraintprogrammierung wurde es durch [DSV90] eingeführt. Seitdem gilt es hier als Benchmark-Problem.

Das Problem besteht darin, die Brücke in Abbildung 3.3 zu bauen. Das Problem ist in Tabelle 3.2 spezifiziert, woraus sich Präzedenz- und Kapazitätsconstraints ergeben (als Zeiteinheit wird ein Tag gewählt). Auch hier nehmen wir an, daß auf einer Ressource nicht mehr als eine Aufgabe gleichzeitig bearbeitet werden kann. Aus der Tabelle ergeben sich z. B. für die Verschalungsar-

**Abbildung 3.3** Die zu bauende Brücke




[2]Kommen für eine Firma die Aufgaben $x_1, x_2, \ldots, x_n$ in dieser Reihenfolge in der Tabelle 3.1 vor, so wird zuerst mit $(\{x_1 + dur(x_1) \leq x_2\},\ \{x_2 + dur(x_2) \leq x_1\})$, dann mit $(\{x_1 + dur(x_1) \leq x_3\},\ \{x_3 + dur(x_3) \leq x_1\})$ etc. distribuiert. Anschließend wird die Distribuierung für die Aufgaben $x_2, \ldots, x_n$ durchgeführt.

beiten folgende Präzedenzconstraints:

$$A1+4 \leq S1, \quad A2+2 \leq S2, \quad P1+20 \leq S3, \quad P2+13 \leq S4, \quad A5+2 \leq S5, \quad A6+5 \leq S6.$$

Weiterhin sind folgende zusätzliche Bedingungen zu erfüllen.

**(1)** Zwischen der Fertigstellung der Verschalung und dem Gießen der Betonfundamente dürfen nicht mehr als 4 Tage vergehen.

**(2)** Zwischen dem Ende der Ausschachtung und dem Beginn der Verschalung dürfen höchstens 3 Tage vergehen.

**(3)** Die Verschalung muß mindestens 6 Tage nach Errichten der Bauhütten beginnen.

**(4)** Der Abbau der Bauhütten kann 2 Tage vor Ende des Mauerns beginnen.

**(5)** Die Auslieferung der Träger ist am 30. Tag nach dem Projektbeginn.

Das Modell des vorangehenden Beispiels kann übernommen werden, außer daß wir für die obere Schranke der Gesamtzeit nun 271 (der Summe der Ausführungszeiten) verwenden und die zusätzlichen Bedingungen modelliert werden müssen. Dies geschieht durch die folgenden Constraints und die entsprechenden Projektoren.

$$\begin{array}{ll}
(1) & (Bi+dur(Bi))-(Si+dur(Si)) \leq 4, \quad 1 \leq i \leq 6 \\
(2) & Si-(Ai+dur(Ai)) \leq 3, \quad i \in \{1,2,5,6\} \\
 & S3-(P1+dur(P1)) \leq 3 \qquad S4-(P2+dur(P2)) \leq 3 \\
(3) & UE+6 \leq Si, \quad 1 \leq i \leq 6 \\
(4) & (Mi+dur(Mi))-2 \leq UA, \quad 1 \leq i \leq 6 \\
(5) & L = PA+30
\end{array}$$

Als Optimalitätskriterium dient die Dauer des Brückenbaus, also der Zeitpunkt des Projektendes. Diesen Zeitpunkt nennen wir die *Länge* eines Schedules (ein Schedule ist der konstruierte Ablaufplan; siehe auch Abschnitt 5.1). Wie im vorangehenden Beispiel soll die optimale Lösung durch Branch-and-Bound gefunden werden.

Bei Benutzung der first-fail Distribuierungsstrategie und von Kapazitätsconstraints ergibt sich als erste Lösung ein Schedule der Länge 130 Tage, wobei der Suchbaum 97 948 Wahlpunkte und 97 922 Fehlerknoten enthält. Sucht man jedoch mit Branch-and-Bound nach der optimalen Lösung des Problems, so ist noch nach über 500 000 Wahlpunkten keine bessere Lösung gefunden worden. Während diese Strategie also für das erste Schedulingbeispiel sehr gut war, versagt sie hier völlig.

Deshalb wählen wir nun die Distribuierungsstrategie, die die Aufgaben auf den Ressourcen serialisiert (bei gleichzeitiger Verwendung von Kapazitätsconstraints). Die Reihenfolge der Serialisierung sei dabei analog zum Vorkommen in Tabelle 3.2 (also zuerst die Aufgaben, die den Bagger benötigen, dann die, die die Ramme benötigen etc.). Auf diese Weise wird die optimale Lösung der Länge 104 nach 91 176 Wahlpunkten gefunden. Jedoch erfordert die Lösung des Gesamtproblems (also inklusive des Beweises der Optimalität) noch immer 335 522 Wahlpunkte, 335 520 Fehlerknoten und 3 Lösungsknoten.

Da dieses Resultat nicht zufriedenstellend ist, wählen wir jetzt eine andere Distribuierungsstrategie. Analog zu first-fail soll zuerst ein Engpaß serialisiert. Ein einfaches Kriterium hierfür ist die Auslastung einer Ressource, also die Summe der Ausführungszeiten der auf ihr ausgeführten Aufgaben. Die Aufgaben auf der Ressource mit der größten Auslastung sollen zuerst serialisiert werden. Für das Brückenproblem werden somit nacheinander die Aufgaben, die Maurer, Kran, Zimmerhandwerker, Ramme, Planierraupe, Bagger und Betonmischer benötigen, serialisiert werden. Die erste Lösung mit einer Länge von 110 Tagen wird mit 93 Wahlpunkten und keinem Fehlerknoten gefunden. Die optimale Lösung wird in einem Suchbaum mit nur noch 436 Wahlpunkten, 434 Fehlerknoten und 3 Lösungsknoten gefunden und ihre Optimalität bewiesen.

In späteren Kapiteln werden wir sehen, wie die Kapazitätsconstraints und die Distribuierungsstrategien weiter verfeinert werden können, so daß selbst sehr schwierige und große Probleme gelöst werden können. Mit den in den Kapiteln 5 und 6 vorgestellten Techniken kann die Zahl der benötigten Wahlpunkte zur Lösung des Brückenbauproblems nochmals um eine Größenordung reduziert werden (siehe Abschnitt 9.4.1).

**Tabelle 3.2** Spezifikation des Brückenbauproblems

| N | Aufg. | Inhalt | Ausf.-Zeit | Vorgänger | Ressource |
|---|---|---|---|---|---|
| 1 | PA | Projektbeginn | 0 | – | – |
| 2 | A1 | Ausschachtung (Außenpfeiler 1) | 4 | PA | Bagger |
| 3 | A2 | Ausschachtung (Pfeiler 1) | 2 | PA | Bagger |
| 4 | A3 | Ausschachtung (Pfeiler 2) | 2 | PA | Bagger |
| 5 | A4 | Ausschachtung (Pfeiler 3) | 2 | PA | Bagger |
| 6 | A5 | Ausschachtung (Pfeiler 4) | 2 | PA | Bagger |
| 7 | A6 | Ausschachtung (Außenpfeiler 2) | 5 | PA | Bagger |
| 8 | P1 | Pfeilerfundamente 3 | 20 | A3 | Ramme |
| 9 | P2 | Pfeilerfundamente 4 | 13 | A4 | Ramme |
| 10 | UE | Errichten der Bauhütten | 10 | PA | – |
| 11 | S1 | Verschalung (Außenpfeiler 1) | 8 | A1 | Zimmerhandwerker |
| 12 | S2 | Verschalung (Pfeiler 1) | 4 | A2 | Zimmerhandwerker |
| 13 | S3 | Verschalung (Pfeiler 2) | 4 | P1 | Zimmerhandwerker |
| 14 | S4 | Verschalung (Pfeiler 3) | 4 | P2 | Zimmerhandwerker |
| 15 | S5 | Verschalung (Pfeiler 4) | 4 | A5 | Zimmerhandwerker |
| 16 | S6 | Verschalung (Außenpfeiler 2) | 10 | A6 | Zimmerhandwerker |
| 17 | B1 | Betonfundamente (Außenpfeiler 1) | 1 | S1 | Betonmischer |
| 18 | B2 | Betonfundamente (Pfeiler 1) | 1 | S2 | Betonmischer |
| 19 | B3 | Betonfundamente (Pfeiler 2) | 1 | S3 | Betonmischer |
| 20 | B4 | Betonfundamente (Pfeiler 3) | 1 | S4 | Betonmischer |
| 21 | B5 | Betonfundamente (Pfeiler 4) | 1 | S5 | Betonmischer |
| 22 | B6 | Betonfundamente (Außenpfeiler 2) | 1 | S6 | Betonmischer |
| 23 | AB1 | Betongießen (Außenpfeiler 1) | 1 | B1 | – |
| 24 | AB2 | Betongießen (Pfeiler 1) | 1 | B2 | – |
| 25 | AB3 | Betongießen (Pfeiler 2) | 1 | B3 | – |
| 26 | AB4 | Betongießen (Pfeiler 3) | 1 | B4 | – |
| 27 | AB5 | Betongießen (Pfeiler 4) | 1 | B5 | – |
| 28 | AB6 | Betongießen (Außenpfeiler 2) | 1 | B6 | – |
| 29 | M1 | Mauern (Außenpfeiler 1) | 16 | AB1 | Maurer |
| 30 | M2 | Mauern (Pfeiler 1) | 8 | AB2 | Maurer |
| 31 | M3 | Mauern (Pfeiler 2) | 8 | AB3 | Maurer |
| 32 | M4 | Mauern (Pfeiler 3) | 8 | AB4 | Maurer |
| 33 | M5 | Mauern (Pfeiler 4) | 8 | AB5 | Maurer |
| 34 | M6 | Mauern (Außenpfeiler 2) | 20 | AB6 | Maurer |
| 35 | L | Auslieferung der Träger | 2 | – | Kran |
| 36 | T1 | Plazierung (Träger 1) | 12 | M1, M2, L | Kran |
| 37 | T2 | Plazierung (Träger 2) | 12 | M2, M3, L | Kran |
| 38 | T3 | Plazierung (Träger 3) | 12 | M3, M4, L | Kran |
| 39 | T4 | Plazierung (Träger 4) | 12 | M4, M5, L | Kran |
| 40 | T5 | Plazierung (Träger 5) | 12 | M5, M6, L | Kran |
| 41 | UA | Entfernung der Bauhütten | 10 | – | – |
| 42 | V1 | Aufschütten 1 | 15 | T1 | Planierraupe |
| 43 | V2 | Aufschütten 2 | 10 | T5 | Planierraupe |
| 44 | PE | Projektende | 0 | T2, T3, T4, V1, V2, UA | – |

# Kapitel 4

# Ausgewählte Projektoren

In den vorangehenden Kapiteln haben wir gesehen, wie kombinatorische Probleme durch das Zusammenwirken von Propagierung und Distribuierung gelöst werden können. Nun stellt sich die Frage, welche Constraints, Projektoren und Distribuierungsstrategien benötigt werden, um möglichst viele Anwendungsprobleme effizient lösen zu können. Es stellt sich also die Frage nach der notwendigen *Expressivität* und *Effizienz*.

Natürlich kann es hierfür kein eindeutiges Kochrezept geben. Neue Anwendungen werden auch immer neue Constraints bzw. Projektoren erfordern und Distribuierungsstrategien werden anwendungsspezifische Charakteristiken berücksichtigen müssen. Doch aus unserer Erfahrung heraus können wir eine Auswahl präsentieren, die sehr häufig benötigt wird (siehe auch zum Beispiel [Van89, VSD92, AB93, BC94, Wal96, ILOG96b]). Wir werden auch eine Möglichkeit aufzeigen, eine Vielzahl von Constraints durch Kombination von einfachen Constraints auszudrücken (sogenannte reifizierte Constraints). Dadurch können bereits vorhandene Projektoren zur Realisierung neuer Constraints wiederverwendet werden. Eine Alternative hierzu ist es, Projektoren in einer Programmiersprache zu entwickeln. Dies wird in Kapitel 7 behandelt.

In den folgenden Abschnitten beschreiben wir zuerst einen Constraint und dann eine oder mehrere Realisierungen durch Projektoren. Für die Projektoren führen wir oft auch eine Komplexitätsanalyse der Laufzeit durch. Hierzu machen wir die folgenden Annahmen. Für die untere Schranke der Laufzeitkomplexität der Propagierungsfunktion $N_p$ eines Projektors $p$, angewendet auf einen Bereichsconstraint $d$, nehmen wir $N_p(d) \neq \bot$ an. Ansonsten könnte die Berechnung von $N_p(d)$ vorzeitig abgebrochen werden und wir hätten oft die triviale untere Schranke $\Omega(1)$ (siehe Abschnitt 4.1). Auch nehmen wir an, daß Operationen auf einem Bereich, wie das Bestimmen des kleinsten Werts des Bereichs, aber auch die Schnittoperation zwischen zwei Bereichen, konstante Zeit benötigen.[1] Beachte, daß wir für Komplexitätsbetrachtungen in diesem wie in späteren Kapiteln annehmen, daß Projektoren nur die Bereiche ihrer Argumente berücksichtigen (siehe auch Abschnitt 2.4).

[1] Falls die Bereiche keine Lücken enthalten (also Intervalle sind) ist die Annahme für Operationen zwischen Bereichen offensichtlich gerechtfertigt. In anderen Fällen hängt eine exakte Analyse von der tatsächlichen Implementierung von Bereichen ab (siehe Abschnitt 11.2). Für praktische Anwendungen zeigt eine empirische Analyse, daß die Operationen auf Bereichen nur einen sehr geringen Teil der Laufzeit ausmachen.

## 4.1 Gleichungen und Ungleichungen

Gleichungen und Ungleichungen kommen in der Modellierung von fast jedem kombinatorischen Problem vor. Sie werden deshalb hier genauer betrachtet. Beispiele sind bereits in den beiden vorangehenden Kapiteln zu finden. Nach unserem Wissen gibt es über zugrundeliegende Propagierungsalgorithmen keine Literatur.

Die Realisierung einer Gleichung bzw. Ungleichung ist nicht kompositional und deshalb sollte eine Gleichung/Ungleichung durch einen einzelnen Projektor realisiert werden. Kommt nämlich in einer Spezifikation eine Gleichung $s = t$ (oder eine Ungleichung) vor und wird die Gleichung in einer zulässigen Konfiguration durch mehrere Projektoren modelliert, so können diese Projektoren zusammen prinzipiell nicht das gleiche Propagierungsverhalten erreichen wie ein einzelner Projektor, der $s = t$ realisiert. Ein Beispiel ist die Gleichung $X + Y - X = Z$. Selbst durch die Modellierung durch zwei vollständige Projektoren für $X + Y = W$ und $W - X = Z$ kann durch die in Kapitel 2 beschriebenen Reduktionsregeln nicht bemerkt werden, daß die Gleichung äquivalent zu $Y = Z$ ist. Dazu betrachten wir den Bereichsconstraint $X \in 3\#6 \wedge Y \in [7\ 9] \wedge Z \in 0\#50$, für den ein vollständiger Projektor für $X + Y - X = Z$ den Bereich von $Z$ auf $[7\ 9]$ reduzieren wird. Führen wir die Hilfsvariable $W$ mit $W \in 0\#50$ ein, so ergeben die Projektoren für $X + Y = W$ und $W - X = Z$ nur den Bereich $4\#12$ für $Z$. Ähnliches gilt für die Ungleichung $X + Y - X \leq Z$ mit dem obigen Bereichsconstraint. In diesem Fall erhält man den Bereich $7\#50$ bzw. $4\#50$ für $Z$.

Wir gehen jetzt auf effiziente Projektoren für Gleichungen und Ungleichungen ein. Dazu betrachten wir den Constraint $c$:

$$a_1 \cdot x_1 + \ldots + a_n \cdot x_n \sim_\rho 0,$$

für ein beliebiges $n \in \mathbb{N}$, wobei $\sim\rho$ für ein Relationssymbol aus $\{=, \leq, <, \geq, >, \neq\}$ steht und die $a_i$ ganze Zahlen sind, $1 \leq i \leq n$. Wir nehmen an, daß die Variablen alle verschieden sind (ansonsten kann man geeignet umformen). Sei $d$ der Bereichsconstraint, auf den die zu beschreibende Reduktionsfunktion $N_p$ des $c$ realisierenden Projektors $p$ angewendet wird.

Dann ist $N_p$ wie folgt definiert. Zuerst wird jedes Produkt $a_k \cdot x_k$ isoliert:

$$a_k \cdot x_k \ \sim_\rho \ \underbrace{-\sum_{i=1, i\neq k}^{n} a_i \cdot x_i}_{RHS_k}.$$

Wir schreiben $\underline{y}$ für $\min(dom(y,d))$ und $\overline{y}$ für $\max(dom(y,d))$. Nun berechnen wir für die Summe $RHS_k$ die obere bzw. unterer Schranke $\overline{RHS_k}$ bzw. $\underline{RHS_k}$:

$$\begin{aligned}
\overline{RHS_k} &= -\sum_{i=1, i\neq k, a_i>0}^{n} a_i * \underline{x}_i - \sum_{i=1, i\neq k, a_i<0}^{n} a_i * \overline{x}_i \\
\underline{RHS_k} &= -\sum_{i=1, i\neq k, a_i>0}^{n} a_i * \overline{x}_i - \sum_{i=1, i\neq k, a_i<0}^{n} a_i * \underline{x}_i
\end{aligned}$$

Diese Werte werden benutzt, um den Bereich von $x_k$ einzuschränken. Die Bereiche werden so lange reduziert, bis keine weiteren Einschränkungen mehr möglich sind (wegen der gewünschten Idempotenz von $N_p$; siehe Seite 20). Wir beschreiben nun die Propagierung für die Constraints $s \sim_\rho 0$, wobei $s$ für $a_1 \cdot x_1 + \ldots + a_n \cdot x_n$ steht.

$s \leq 0$**:** Die Reduktion der Bereiche folgt den folgenden Ungleichungen.

$$\begin{aligned} x_k &\leq \left\lfloor \frac{\overline{RHS_k}}{a_k} \right\rfloor \quad \text{falls } a_k > 0 \\ x_k &\geq \left\lceil \frac{\overline{RHS_k}}{a_k} \right\rceil \quad \text{falls } a_k < 0 \end{aligned} \tag{4.1}$$

Sei $d'$ der aus diesen Ungleichungen resultierende Bereichsconstraint. Es gilt $dom(x_k, d') = dom(x_k, d) \cap \min(dom(x_k, d))\#m$ für die Schreibweise $x_k \leq m$ und $dom(x_k, d') = dom(x_k, d) \cap m\#\max(dom(x_k, d))$ für die Schreibweise $x_k \geq m$. Würde auf diese Weise ein leerer Bereich entstehen, wird die Berechnung abgebrochen und es gilt $N_p(d) = \bot$.

Für den Projektor gilt $E_p(d) = 1$, wenn die folgende Bedingung gilt.

$$\sum_{i=1, a_i>0}^{n} a_i * \overline{x}_i \; + \sum_{i=1, a_i<0}^{n} a_i * \underline{x}_i \leq 0$$

Der Projektor ist vollständig. Als ein Beispiel betrachten wir

$$X - Y \leq Z - V$$

Wir haben die folgenden Einschränkungen.

$$X \leq \overline{Z} - \underline{V} + \overline{Y} \qquad Y \geq \underline{X} - \overline{Z} + \overline{V} \qquad Z \geq \underline{X} - \overline{Y} + \underline{V} \qquad V \leq \overline{Z} - \underline{X} + \overline{Y}$$

Der Projektor signalisiert Subsumtion, wenn $\overline{X} - \underline{Y} \leq \underline{Z} - \overline{V}$ gilt.

$s \geq 0$**:** Dieser Fall kann zu dem Fall $s \leq 0$ reduziert werden, da

$$a_1 \cdot x_1 + \ldots + a_n \cdot x_n \; \geq \; 0$$

äquivalent zu

$$(-a_1) \cdot x_1 + \ldots + (-a_n) \cdot x_n \; \leq \; 0$$

ist. Alternativ kann $\underline{RHS_k}$ zur Definition benutzt werden.

$s < 0$**:** Analog zu $s \leq 0$.

$s > 0$**:** Analog zu $s \geq 0$.

$s = 0$**:** In diesem Fall ist $N_p$ sowohl durch die Reduktionsregeln für $s \leq 0$ als auch für $s \geq 0$ definiert. Der Projektor ist somit nicht reduktionsvollständig, da z. B. für $2 \cdot Y = X$ die ungeraden Werte aus dem Inneren des Bereichs von $X$ nicht entfernt werden. Für den Projektor gilt $E_p(d) = 1$, falls $d$ determiniert und die Gleichung erfüllt ist. Damit ist der Projektor subsumtionsvollständig.

Somit implementiert dieser Projektor ein schwächeres Propagierungsverhalten als zum Beispiel Intervallkonsistenz [VSD95]. Sei dazu ein Constraint $c$ und ein Bereichsconstraint $x_1 \in \delta_1 \wedge \ldots \wedge x_n \in \delta_n$ gegeben. Dann heißt *c intervallkonsistent* genau dann, wenn für alle $i \in \{1, \ldots, n\}$ und $v_i \in \{\min(\delta_i), \max(\delta_i)\}$ es Werte $v_j \in \delta_j, j \neq i$, gibt, so daß $(v_1, \ldots, v_n) \in \rho(c)$ gilt. Zum Beispiel ist $X \in 2\#10 \wedge Y \in 4\#10 \wedge Z \in 2\#6$ mit dem Constraint $3 \cdot X = 5 \cdot Y - 7 \cdot Z$ nicht intervallkonsistent (für $X = 10$ existiert keine Lösung). Jedoch kann die vorgestellte Realisierung der Gleichung die Bereiche nicht weiter reduzieren und die Inkonsistenz nicht erkennen.

$s \neq 0$**:** Wenn noch höchstens eine Variable $x_k$ nicht determiniert ist, dann bezeichnet $RHS_k$ eine eindeutige ganze Zahl. Diese Zahl wird dann aus dem Bereich von $x_k$ entfernt. Für den Projektor gilt $E_p(d) = 1$, falls $d$ determiniert und die Ungleichung erfüllt ist. Damit ist der Projektor zwar reduktions- aber nicht subsumtionsvollständig (da zum Beispiel $X \in 1\#2 \wedge Y \in 5\#7 \models X \neq Y$ nicht erkannt wird).

Ungleichungen mit den Relationssymbolen $<, \leq, >$ oder $\geq$ werden mit den vorgestellten Algorithmen durch vollständige Projektoren realisiert. Um den Bereich jeder Variablen einmal einzuschränken müssen $n$ Summen berechnet werden, die jeweils $n-1$ Summanden haben. Wird in (4.1) die Ungleichung $x_i \leq m$ für eine Variable $x_i$ angewendet, so kommt nur $\underline{x_i}$ in $\overline{RHS_k}$ oder in $\underline{RHS_k}$ vor (und entsprechend für $x_i \geq m$). Somit ist die Laufzeitkomplexität quadratisch in der Zahl der vorkommenden Variablen, also $\Theta(n^2)$, wobei wir annehmen, daß das Aufsummieren von $n$ Zahlen eine Komplexität von $O(n)$ besitzt. Der Projektor für $s \neq 0$ hat offensichtlich eine Laufzeitkomplexität von $\Theta(n)$.

Sei $k = \max(\{|dom(x_1,d)|, \ldots, |dom(x_n,d)|\})$, also die maximale Kardinalität eines Bereichs einer in $s = 0$ vorkommenden Variablen. Eine obere Schranke für die Laufzeitkomplexität des beschriebenen Projektors für Gleichungen ist $O(n^3 \cdot k)$, da zur Reduktion der Bereiche von $x_1$ bis $x_n$ jedesmal maximal $2 \cdot n$ Summenformeln berechnet werden müssen und evtl. nur jedesmal ein Element aus einem Bereich entfernt wird (und es $n \cdot k$ Werte in den Bereichen geben kann). In realistischen Anwendungen ist die Laufzeit eines Gleichungsprojektors aber oft kleiner, da in den meisten Fällen der Fixpunkt der Berechnung früher erreicht wird. Eine untere Schranke für die Laufzeitkomplexität ist $\Omega(n^2)$. Nur in Fällen wie $2 \cdot X = 2 \cdot Y + 1$ wird die schlechteste Laufzeit wirklich erreicht.

Ein vollständiger Projektor für eine Gleichung hätte hingegen die Komplexität $O(n \cdot k^n)$. Zwar kann ein vollständiger Projektor den Suchraum stärker einschränken als ein unvollständiger Projektor, doch die Beispiele in Kapitel 3 zeigen deutlich, daß die größere Komplexität diesen Vorteil in der Regel zunichte macht.

Neben den linearen Gleichungen und Ungleichungen, wie sie oben beschrieben wurden, werden in Anwendungen oft auch nicht-lineare Gleichungen und Ungleichungen benötigt (siehe z. B. Abschnitt 3.1.1). Dabei bedeutet nicht-linear, daß auch Produkte von Variablen erlaubt sind. Hier kann in der Regel jedoch nicht nach einer Variablen aufgelöst werden, wie es im linearen Fall geschieht (siehe zum Beispiel $X \cdot Y + Y^2 = Z$). Stattdessen kann man zum Beispiel jede Variablenposition isolieren und dadurch Propagierungsalgorithmen ableiten (für das Beispiel ergibt sich $Y = (z - Y^2)/X$ sowie $Y = (z - X \cdot Y)/Y$ etc.).

## 4.2 Symbolische und globale Constraints

Neben den grundlegenden arithmetischen Constraints wie Gleichungen gibt es auch Constraints, die durch beliebige Verknüpfung aus anderen Constraints hervorgehen. So kann zum Beispiel der Constraint, der besagt, daß $n$ Variablen $x_1$ bis $x_n$ paarweise verschiedene Werte annehmen sollen, als Konjunktion von $n \cdot (n-1)/2$ Constraints $x_i \neq x_j$ für $1 \leq i < j \leq n$ geschrieben werden. Für solche Constraints, die aus vielen arithmetischen Constraints zusammengesetzt werden können, haben sich die Begriffe *symbolische* oder *globale* Constraints etabliert [DVS+88, AB93].

Die grundlegende Idee bei symbolischen Constraints ist es nun, den Constraint nicht durch eine Menge von Projektoren zu realisieren, sondern möglichst durch einen einzigen Projektor. Dieser einzelne Projektor kann dann zu mehr Propagierung führen als mehrere verschiedene Projektoren (und ist in einer Implementierung auch effizienter auszuführen als mehrere verschiedene; siehe auch Kapitel 11). Während die vielen Projektoren nämlich prinzipiell nur jeweils wenige Variablenbereiche beeinflussen können, hat ein einzelner Projektor, der den symbolischen Constraint realisiert, eine globale Sicht auf mehr Variablen und deren Bereiche. Somit können stärkere Propagierungsalgorithmen realisiert werden. In gewisser Weise wird somit auch ein Schritt von lokaler Konsistenz hin zu globaler Konsistenz getan (siehe Kapitel 2). Außerdem erlauben symbolische Constraints in einer konkreten Implementierung eine natürliche und kompakte Modellierung.

Wir unterscheiden zwischen allgemeinen symbolischen oder globalen Constraints und solchen, die speziell für bestimmte Anwendungen entwickelt wurden. Letztere werden in Abschnitt 4.3 besprochen. Eine scharfe Trennung ist hier jedoch nicht möglich.

Den Constraint, daß $n$ Variablen paarweise verschiedene Werte annehmen sollen, haben wir schon früher kennengelernt. Ein möglicher Projektor entfernt die Werte aller determinierten Variablen aus den Bereichen der noch nicht determinierten Variablen. Für die meisten betrachteten Anwendungen ist diese Propagierungsstärke ausreichend (siehe zum Beispiel Kapitel 3). Der Projektor hat eine Zeitkomplexität von $O(n^2)$, da bis zum Erreichen eines Fixpunktes gerechnet werden muß (wegen der geforderten Idempotenz). Während ein naiver vollständiger Algorithmus exponentielle Komplexität hat, gibt es jedoch auch eine vollständige Variante mit der Komplexität $O(n^2 \cdot k^2)$, wenn $k$ die maximale Kardinalität eines Bereiches ist (siehe [Reg94]). Ein Projektor, der das Propagierungsverhalten des ersten Projektors leicht verstärkt und noch immer effizient ist, wird berücksichtigen, daß die Anzahl der insgesamt möglichen Werte (also $|\delta_1 \cup \ldots \cup \delta_n|$ für einen Bereichsconstraint $x_1 \in \delta_1 \wedge \ldots \wedge x_n \in \delta_n$) immer größer oder gleich sein muß als die Anzahl der Variablen, also $n$.

Ein weiterer typischer symbolischer Constraint besagt, daß genau $y$ Elemente in einer Liste von $x_1$ bis $x_n$ gleich der ganzen Zahl $v$ sein sollen. Ein vollständiger Projektor für diesen Constraint hat die Zeitkomplexität $\Theta(n)$, da für jede Variable $x_1$ bis $x_n$ geprüft werden muß, ob $v$ in dem zugehörigen Bereich enthalten ist. Entsprechend gibt es Constraints, die besagen, daß die Anzahl der Elemente in der Liste, die gleich $v$ sind, kleiner oder größer als $y$ sein sollen.

Weitere Beispiele für symbolische Constraints sind zum Beispiel in [RP97, CL97b, BC97] zu finden.

## 4.3 Anwendungsspezifische Constraints

Ein Beispiel für einen anwendungsspezifischen globalen Constraint haben wir bereits in Kapitel 3 kennengelernt, nämlich Kapazitätsconstraints für Schedulinganwendungen. Diese besagen, daß auf einer Ressource nicht mehr Aufgaben gleichzeitig bearbeitet werden können, als es die Kapazität der Ressource zuläßt. Solche Constraints und deren Realisierung durch Projektoren sind Gegenstand von Kapitel 5. Anwendungen finden sich in den Kapiteln 8 bis 10.

Ein weiterer häufig verwendeter anwendungsspezifischer Constraint ist derjenige, der besagt,

daß eine Menge von $n$ Rechtecken paarweise nicht überlappen darf. Dazu wird die Position jedes Rechtecks durch zwei Variablen $x_i$ und $y_i$ charakterisiert, die die Koordinaten der unteren linken Ecke des Rechtecks angeben. Außerdem werden je zwei ganze Zahlen $w_i$ und $h_i$ mit einem Rechteck assoziiert, nämlich seine Breite und seine Höhe. Der angegebene Constraint läßt sich somit durch die Formel

$$\forall i \in \{1,\ldots,n-1\} \forall j \in \{i+1,\ldots,n\} : \quad \begin{array}{l} x_i + w_i \leq x_j \ \vee \ x_j + w_j \leq x_i \\ \vee \ y_i + h_i \leq y_j \ \vee \ y_j + h_j \leq y_i \end{array}$$

spezifizieren. Die beiden ersten Klauseln jeder Disjunktion beschreiben den Fall, daß sich zwei Rechtecke nicht in $x$-Richtung überlappen. Die beiden letzten Klauseln decken die $y$-Richtung ab. Projektoren, die diesen Constraint realisieren, können offensichtlich stark in ihrer Propagierungsstärke und Effizienz variieren. Ein effizienter Projektor wird genau das gleiche Propagierungsverhalten haben wie eine Menge von Projektoren, in der jeder Projektor genau eine der obigen Disjunktionen realisiert (z. B. durch reifizierte Constraints; siehe den folgenden Abschnitt). Ein berechnungsintensiverer Projektor mit stärkerer Propagierung wird ähnliche Techniken verwenden wie sie in Kapitel 5 beschrieben werden (siehe auch [BC94]). Beachte, daß das durch den Constraint beschriebene Problem NP-vollständig ist (da Kapazitätsconstraints für disjunktive Schedulingprobleme ein Spezialfall hiervon sind; siehe Abschnitt 5.1). Dieser Constraint kann offensichtlich für Plazierungsprobleme verwendet werden.

Weitere anwendungsspezifische globale Constraints sind zum Beispiel in [AB93, BC94] oder [CL96c] zu finden.

## 4.4 Reifizierte Constraints

Trotz der Vielfalt der bisher vorgestellten Constraints und deren Projektoren, wird es immer Bedarf für neue und oft benutzerspezifische Constraints und Projektoren geben. Wir betrachten hier eine spezielle Methode, um Constraints über den Wahrheitswerten von Constraints zu realisieren. Auf diese Art lassen sich auch viele symbolische Constraints realisieren (bevor diese zum Beispiel effizienter als einzelner Projektor zur Verfügung stehen).

*Reifikation* eines Constraints $c$ bedeutet, daß seine Gültigkeit in eine Variable $x$ reflektiert wird, für die der Bereichsconstraint $x \in 0\#1$ gilt (solche Variablen nennen wir *0/1-Variablen*):

$$c \leftrightarrow x = 1.$$

Reifizierte Constraints wurden erstmals in [OB93] eingeführt. Der Begriff geht auf G. Smolka zurück und wurde in [HW96b] geprägt. Die Idee, Constraints über dem Wahrheitswert anderer Constraints zu formulieren, findet sich auch in [VD91b]. Ähnliche Ideen gibt es auch im Operations Research (siehe zum Beispiel [NW88]), um disjunktive Constraints zu formulieren.

Sei $p$ der Projektor, der $c$ realisiert, sei $r$ der Projektor, der den reifizierten Constraint $c \leftrightarrow x = 1$ realisiert und sei $d$ der Bereichsconstraint $x \in \delta_x \ \wedge \ \ldots$. Wir definieren nun die Propagierungsfunktion $N_r$ und die Subsumtionsfunktion $E_r$ des Projektors $r$. Diese Funktionen hängen von den entsprechenden Funktionen für $p$ und $q$ ab, wobei $q$ den Constraint $\neg c$ realisiert. In Abhängigkeit von den verwendeten Projektoren kann ein Constraint auf verschiedene Arten reifiziert werden.

- Gilt $dom(x,d) = \{1\}$, so gilt $N_r(d) = N_p(d)$ und $E_r(d) = E_p(d)$.
- Gilt $dom(x,d) = \{0\}$, so gilt $N_r(d) = N_q(d)$ und $E_r(d) = E_q(d)$.
- Gilt $E_p(d) = 1$ und $1 \in dom(x,d)$, so gilt $N_r(d) = (x = 1 \ \wedge \ \ldots)$ und $E_r(d) = 1$. Gilt hingegen $E_p(d) = 1$ und $1 \notin dom(x,d)$, so gilt $N_r(d) = \bot$.
- Gilt $N_p(d) = \bot$ und $0 \in dom(x,d)$, so gilt $N_r(d) = (x = 0 \ \wedge \ \ldots)$ und $E_r(d) = 1$. Gilt hingegen $N_p(d) = \bot$ und $0 \notin dom(x,d)$, so gilt $N_r(d) = \bot$.
- In allen anderen Fällen gilt $N_r(d) = d$.

Eine alternative Formulierung könnte im vierten Fall $E_q(d) = 1$ in der Vorbedingung verwenden. Der Vorteil der angeführten Formulierung liegt darin, daß zur Implementierung der Projektor $p$ ausreicht, so lange nicht der Fall $dom(x,d) = \{0\}$ eintritt.

Damit läßt sich nun leicht zum Beispiel eine Disjunktion ausdrücken. So kann die Spezifikation $(d, \{c_1 \vee c_2\})$ durch die Konfiguration $(d, \{c_1 \leftrightarrow x_1 = 1,\ c_2 \leftrightarrow x_2 = 1,\ x_1 + x_2 \geq 1\})$ realisiert werden. Stellt der Projektor $p_{c_1}$ Dissubsumtion von $c_1$ fest (also $N_{p_{c_1}}(d) = \bot$), so wird $x_1$ zu 0 determiniert. Wegen $x_1 + x_2 \geq 1$ wird dann aber $x_2$ zu 1 determiniert. Damit wird der Projektor des reifizierten Constraints für $c_2 \leftrightarrow x_2 = 1$ das Propagierungsverhalten von $c_2$ simulieren.

Als weiteres Beispiel wählen wir den symbolischen Constraint, daß unter den Variablen $x_1$ bis $x_n$ genau $y$ den Wert $v$ annehmen (siehe Abschnitt 4.2). Dieser Constraint kann in einer Konfiguration durch die Projektoren $(x_1 = v) \leftrightarrow (y_1 = 1)$ bis $(x_n = v) \leftrightarrow (y_n = 1)$ und $y_1 + \ldots y_n = y$ realisiert werden. Als konkretes Beispiel nehmen wir $v = 3$, $y = 1$ und die drei Variablen $A, B, C$ mit jeweils dem Bereich 0#5 an. Gilt zum Beispiel $A = 3$, so wird $y_A$ aus $(A = v) \leftrightarrow (y_A = 1)$ zu 1 determiniert. Wegen $y_A + y_B + y_C = y$ werden dann aber $y_B$ und $y_C$ zu 0 determiniert. Dies hat wiederum zur Folge, daß aus den Bereichen von $B$ und $C$ der Wert 3 entfernt wird.

## 4.5 0/1-Constraints

Bildet man die Zahlen 0 und 1 auf die Wahrheitswerte *falsch* und *wahr* ab, so kann man logische Verknüpfungen zwischen Variablen ausdrücken. Die üblichen Verknüpfungen wie Konjunktion, Äquivalenz oder Negation sollten als vollständige Projektoren realisiert sein. Ein Beispiel ist $X \wedge Y \equiv Z$. Gilt zum Beispiel $Y = 0$, so kann $Z$ zu 0 determiniert werden. Ist $Z$ gleich 1, so können sowohl $X$ als auch $Y$ zu 1 determiniert werden. Zwar könnten diese Constraints auch durch eine Kombination von reifizierten Constraints modelliert werden, aber die Realisierung durch einen einzelnen Projektor ist effizienter.

Eine Anwendung ist der Constraint, daß eine Variable $X$ größer als 3 sein soll, falls die Variable $Y$ ungleich 4 ist. Dies kann durch die reifizierten Constraints $(X > 3) \leftrightarrow (A = 1)$ und $(Y \neq 4) \leftrightarrow (B = 1)$, sowie den 0/1-Constraint $(B \rightarrow A) \equiv 1$ beschrieben werden.

# Kapitel 5

# Kapazitätsconstraints

Bereits im vorangehenden Kapitel haben wir einige anwendungsspezifische Constraints vorgestellt. In diesem Kapitel stellen wir nun einen Constraint vor, der in Schedulinganwendungen verwendet wird.

*Scheduling* (oder auch *Ablaufplanung*) bedeutet die Zuordnung von Ressourcen zu einer Menge von Aufgaben und das Erstellen eines Zeitplans für die Aufgaben, die um diese Ressourcen konkurrieren. Vielen Schedulingproblemen ist der Constraint gemeinsam, daß zu keinem Zeitpunkt die auf einer Ressource ausgeführten Aufgaben mehr von dieser Ressource verbrauchen dürfen, als es die Ressourcenkapazität zuläßt. Ein solcher Constraint für eine Ressource heißt *Kapazitätsconstraint*.

Wie viele anwendungsspezifische Constraints lassen sich auch Kapazitätsconstraints durch eine Kombination von allgemeinen Projektoren realisieren (nämlich durch reifizierte Constraints; siehe Abschnitt 5.2.1). Wird der Kapazitätsconstraint hingegen durch einen einzelnen Projektor realisiert, ergeben sich zwei Vorteile. Erstens wird in einer konkreten Implementierung Speicherplatz und Laufzeit gespart, da schließlich nur ein einzelner Projektor ausgeführt werden muß. Zweitens, und das steht in diesem Kapitel im Vordergrund, können in diesem Fall mächtige anwendungsspezifische Techniken zur Realisierung einer Propagierungsfunktion des Projektors ausgenutzt werden, so daß stärkere Propagierung resultiert.

Zur Realisierung von Kapazitätsconstraints wollen wir hier die besten verfügbaren Algorithmen verwenden. Die zur Zeit besten Techniken stammen aus dem Operations Research [CP89, AC91, MS96] und sind teilweise in constraintbasierte Ansätze übernommen worden [CL94a, Nui94]. In diesem Kapitel werden wir zeigen, wie diese Techniken in das vorgestellte Modell von Propagierung und Distribuierung eingebettet werden können. Dazu definieren wir geeignete Propagierungsfunktionen, für die wir die Eigenschaften aus Definition 4 zu prüfen haben. Wir können zeigen, daß für einige der in der Literatur angegebenen Verfahren die Monotonie der Propagierungsfunktion nicht garantiert werden kann. Wie solche Propagierungsfunktionen trotzdem für eine Implementierung verwendet werden können, ist in Kapitel 7 und 11 beschrieben. Wir werden jedoch auch zeigen, wie die vorgestellten Algorithmen so angepaßt werden können, daß die resultierenden Propagierungsfunktionen monoton sind.

In Abschnitt 5.1 definieren wir die hier betrachteten Schedulingprobleme und geben an, wie sie

durch geeignete Spezifikationen, wie wir sie aus Kapitel 2 kennen, beschrieben werden können. Wir unterscheiden prinzipiell zwischen *disjunktiven Schedulingproblemen*, für die auf einer Ressource nicht mehr als eine Aufgabe gleichzeitig bearbeitet werden kann, und *kumulativen Schedulingproblemen*, für die eine Ressource gleichzeitig mehrere Aufgaben bearbeiten kann.

In Abschnitt 5.2 schildern wir, wie Kapazitätsconstraints für disjunktive Schedulingprobleme realisiert werden können. Nachdem die grundlegende Technik (genannt *Edge-Finding*) für das Konzept von Propagierung und Distribuierung geeignet aufbereitet wurde, stellen wir zwei konkrete Algorithmen vor, die zur Implementierung von Projektoren verwendet werden können. Der erste Algorithmus ist eine Anpassung eines Verfahrens aus [CL94a] für unser Modell. Der zweite Algorithmus stellt die Übertragung des in [MS96] vorgestellten Verfahrens in die Constraintprogrammierung dar. Wir werden die Propagierung gegenüber dem Verfahren aus [MS96] sogar noch verstärken. Für beide Algorithmen untersuchen wir deren Eignung für den in Kapitel 2 vorgestellten Rahmen und zeigen erstmals, welche Teile der Algorithmen monotone Propagierungsfunktionen definieren.

In Abschnitt 5.3 gehen wir auf kumulative Schedulingprobleme ein. Dazu verallgemeinern wir das aus [MS96] stammende Verfahren für disjunktive Probleme auf kumulative Probleme. Dieses Verfahren wird durch eine weitere von uns entwickelte Propagierungstechnik ergänzt. Das Kapitel endet mit einem Abschnitt über verwandte Arbeiten.

Ein quantitativer Vergleich der hier vorgestellten Realisierungen für eine Reihe von disjunktiven Standardproblemen aus dem Operations Research findet sich in Abschnitt 9.4.1. Ein Vergleich der hier vorgestellten Techniken in einer konkreten Implementierung mit anderen existierenden Systemen findet sich in Abschnitt 13.3. Ein Vergleich der kumulativen Schedulingtechniken wird nicht vorgenommen werden, da starke Distribuierungsstrategien hierzu noch nicht in Oz verfügbar sind.

## 5.1 Scheduling

In diesem Abschnitt charakterisieren wir die Schedulingprobleme, die in dieser Arbeit betrachtet werden. Einen allgemeinen Überblick über das Gebiet bieten zum Beispiel [Bak74, BESW94, ZF94]. Die meisten der in diesem Kapitel verwendeten Begriffe sind Standardbegriffe aus diesem Gebiet. Wir teilen Schedulingprobleme in drei Klassen auf (eine ausführlichere Darstellung dieser Klassen findet sich in [ILOG96a]).

- In einem *reinen Schedulingproblem* ist ein Zeitplan für Aufgaben zu erstellen, die um eine gegebene Menge von Ressourcen konkurrieren. Die Zuordnung einer Aufgabe zu genau einer Ressource ist vorgegeben.
- In einem *reinen Ressourcenzuteilungsproblem* sind Ressourcen einer Menge von gegebenen Aufgaben zuzuordnen. Die Anfangszeiten der Aufgaben sind vorgegeben.
- *Gemischte Scheduling- und Zuteilungsprobleme* zeichnen sich durch eine Mischung der Problemstellungen aus.

In dieser Arbeit werden Beispiele für reine Schedulingprobleme in Kapitel 9 und 10 und für gemischte Probleme in Kapitel 8 betrachtet. In diesem Kapitel betrachten wir nur reine Schedulingprobleme. Wir machen folgende Annahmen. Ein Zeitplan ist zwischen dem Zeitpunkt Null und einer gegebenen oberen Schranke, dem *Schedulinghorizont*, zu erstellen. Jede Aufgabe benötigt genau eine Ressource und wird ohne Unterbrechnung ausgeführt, es ist also *keine Verdrängung* erlaubt (engl. *no preemption*). Die folgende Definition ist an [Nui94] angelehnt.

**Definition 12** *Ein reines* Schedulingproblem *ist wie folgt definiert. $T$ ist eine Menge von Aufgaben, $R$ ist eine Menge von Ressourcen und $H$ ist der Schedulinghorizont. Wir schreiben $t$ für eine Aufgabe in $T$ und $r$ für eine Ressource in $R$. Es sind außerdem die folgenden Funktionen auf $T$ und $R$ gegeben.*

| | |
|---|---|
| $res : T \to R$ | *$res(t)$ ist die Ressource, auf der eine Aufgabe $t$ ausgeführt wird* |
| $cap : R \to \mathbb{N}^+$ | *$cap(r)$ ist die Kapazität einer Ressource $r$ (engl. capacity)* |
| $dur : T \to \mathbb{N}$ | *$dur(t)$ ist die Ausführungszeit einer Aufgabe $t$ (engl. duration)* |
| $use : T \to \mathbb{N}$ | *$use(t)$ ist der Ressourcenverbrauch einer Aufgabe $t$ (engl. usage)* |
| $size : T \to \mathbb{N}$ | *$size(t)$ ist die Größe einer Aufgabe $t$ (engl. size), $size(t) = dur(t) \cdot use(t)$* |
| $util : R \to \mathbb{N}$ | *$util(r)$ ist die Anzahl der Aufgaben, die die Ressource $r$ benötigen (engl. utilization), $util(r) = \lvert\{t \in T \mid res(t) = r\}\rvert$* |

*Für jede Aufgabe $t \in T$ ist eine Startzeit aus $\mathbb{N}$ zu finden, so daß zu keinem Zeitpunkt die auf einer Ressource $r \in R$ ausgeführten Aufgaben mehr von der Ressource verbrauchen, als es die Kapazität $cap(r)$ zuläßt. Zusätzlich können eine Reihe weiterer Bedingungen gegeben sein.*

Wir nehmen im folgenden ein Schedulingproblem mit Mengen $T$ und $R$, einen Schedulinghorizont $H$ und den entsprechenden Funktionen als gegeben an. Die folgende Definition verbindet ein Schedulingproblem mit einer geeigneten Spezifikation.

**Definition 13** *Für ein Schedulingproblem definieren wir wie folgt eine Spezifikation $(d, C)$ aus einem Bereichsconstraint $d$ und einer Menge von Constraints $C$. Wir führen für jede Aufgabe $t \in T$ zwei Variablen $start(t)$ und $compl(t)$ ein, die die* Startzeit *einer Aufgabe bzw. ihre* Fertigstellungszeit *(engl. completion time) bezeichnen. Eine Aufgabe $t$ verbraucht ihre Ressource $res(t)$ zwischen den Zeitpunkten $start(t)$ und $compl(t) - 1$. Wir nehmen an, daß alle zusätzlichen Bedingungen des Schedulingproblems über den Aufgaben in $T$ definiert sind. Der Bereichsconstraint $d$ enthält für eine Variable $start(t)$ den Bereich $0\#(H - dur(t))$ und für die Variable $compl(t)$ den Bereich $dur(t)\#H$.*

*Die Menge $C$ enthält für jede Aufgabe $t$ den Constraint $start(t) + dur(t) = compl(t)$, um Start- und Fertigstellungszeiten zu verbinden. $C$ enthält weiterhin für jede Ressource $r \in R$ den Constraint*

$$\forall i \in \{0, \ldots, H\} : \sum_{t \in T,\, res(t)=r,\, start(t) \le i < compl(t)} use(t) \le cap(r). \tag{5.1}$$

*Der Constraint (5.1) heißt* Kapazitätsconstraint *für eine Ressource $r$. Außerdem enthält $C$ Constraints für die zusätzlichen Bedingungen des Schedulingproblems.*

Das bedeutet, daß in jedem Schedulingproblem eine Menge von Kapazitätsconstraints zu lösen ist. Ein Kapazitätsconstraint besagt, daß zu keinem Zeitpunkt der Ressourcenverbrauch von Aufgaben auf einer Ressource die Kapazität dieser Ressource übersteigen darf.

**Definition 14** *Gegeben sei ein Schedulingproblem P und eine zugehörige Spezifikation* $(d,C)$. *Ein* Schedule *(oder* Zeitplan*) ist eine Belegung, die eine Lösung von* $(d,C)$ *ist. Die* Länge makespan *eines Schedules ist die maximale Fertigstellungszeit einer Aufgabe* $t \in T$ *bzgl. P, also* $makespan = \max(\{compl(t) | t \in T\})$.

Wir verbinden mit einem Schedulingproblem im folgenden immer eine Spezifikation $S = (d,C)$ nach Definition 13. Wir sagen, daß das Schedulingproblem keine Lösung hat, wenn $S$ keine Lösung hat. Wir definieren nun die folgenden Abkürzungen ($d$ ist der Bereichsconstraint aus der Spezifikation).

**Definition 15**

| | |
|---|---|
| $est(t) = \min(dom(start(t), d))$ | *die frühest mögliche Startzeit von t (engl. earliest possible start time)* |
| $lst(t) = \max(dom(start(t), d))$ | *die spätest mögliche Startzeit von t (engl. latest possible start time)* |
| $ect(t) = \min(dom(compl(t), d))$ | *die frühest mögliche Fertigstellungszeit von t (engl. earliest possible completion time)* |
| $lct(t) = \max(dom(compl(t), d))$ | *die spätest mögliche Fertigstellungszeit von t (engl. latest possible completion time)* |

Wir sagen $est(t)$ bzgl. $d$ (und analog für $lst(t)$ etc.), falls der jeweilige Bereichsconstraint $d$ explizit gemacht werden soll. In der Literatur wird oft auch der Begriff *release date* für *est* und *due date* für *lct* verwendet (siehe z. B. [Bak74] oder [BESW94]).

Wir werden in den folgenden Abschnitten verschiedene Realisierungen von Kapazitätsconstraints angeben. Dabei werden wir in einem Bereichsconstraint nur die Variablen für die Startzeiten von Aufgaben $t$ aufschreiben, da die Fertigstellungszeiten durch Constraints $start(t) + dur(t) = compl(t)$ gekoppelt sind. Dabei nehmen wir an, daß der Constraint $start(t) + dur(t) = compl(t)$ durch einen Projektor wie in Abschnitt 4.1 beschrieben realisiert ist; es werden also aus Effizienzgründen keine Lücken in Bereichen propagiert.

Das Problem, ob ein Kapazitätsconstraint bzgl. eines Bereichsconstraints eine Lösung hat, ist NP-vollständig. Damit kann es keinen effizienten vollständigen Projektor für einen Kapazitätsconstraint geben. Der Beweis folgt direkt aus der NP-Vollständigkeit von Schedulingproblemen mit einer Ressource der Kapazität Eins und beliebigen release und due dates (siehe zum Beispiel Seite 236 in [GJ79] oder [BESW94]).

## 5.2 Disjunktive Schedulingprobleme

In diesem Abschnitt beschäftigen wir uns mit Schedulingproblemen, für die auf einer Ressource nicht mehr als eine Aufgabe gleichzeitig bearbeitet werden kann.

**Definition 16** *Eine Ressource r heißt* disjunktiv, *falls* $cap(r) = 1$ *gilt. Ein Schedulingproblem heißt* disjunktiv, *falls alle Ressourcen disjunktiv sind und für alle Aufgaben t gilt:* $use(t) = 1$. *Für disjunktive Schedulingprobleme gilt somit* $size(t) = dur(t)$.

Dies bedeutet, daß für disjunktive Schedulingprobleme jeweils zwei Aufgaben $t_1$ und $t_2$ mit $res(t_1) = res(t_2)$ sich zeitlich nicht überlappen dürfen, also entweder $t_1$ vor $t_2$ fertiggestellt sein muß oder umgekehrt. Damit muß die Disjunktion

$$compl(t_1) \leq start(t_2) \ \vee \ compl(t_2) \leq start(t_1)$$

gelten. Eine alternative Problemformulierung eines Kapazitätsconstraints für disjunktive Schedulingprobleme ist somit, daß die Aufgaben auf einer Ressource *serialisiert* werden müssen. Dazu müssen für $n$ Aufgaben bis zu $O(n^2)$ Ordnungsentscheidungen getroffen werden (siehe das folgende Kapitel über Distribuierungsstrategien). Der Kapazitätsconstraint (5.1) für eine Ressource $r$ ist damit für disjunktive Probleme äquivalent zu

$$\bigwedge_{t_1,t_2 \in T,\, t_1 \neq t_2,\, res(t_1)=res(t_2)=r} compl(t_1) \leq start(t_2) \ \vee \ compl(t_2) \leq start(t_1) \tag{5.2}$$

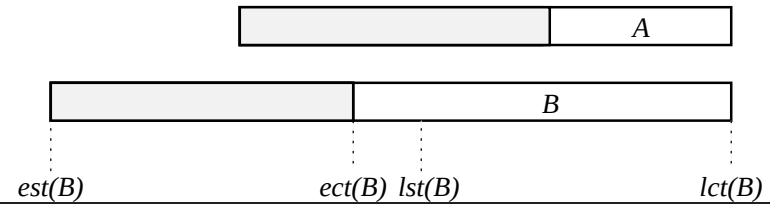
### 5.2.1 Reifizierte Constraints

Eine naheliegende Art, Kapazitätsconstraints zu realisieren, besteht darin, sie durch vollständige Projektoren für reifizierte Constraints zu realisieren (siehe Abschnitt 4.4). Dazu wird eine der Disjunktionen in Formel (5.2) wie folgt modelliert.

$$\begin{gathered}(compl(t_1) \leq start(t_2)) \leftrightarrow x_1 = 1 \\ (compl(t_2) \leq start(t_1)) \leftrightarrow x_2 = 1 \\ x_1 + x_2 = 1\end{gathered}$$

Für ein besseres Verständnis, rechnen wir nun ein Beispiel. Wir betrachten die zwei Aufgaben $A$ und $B$ mit dem Bereichsconstraint $d = start(A) \in 10\#15 \wedge start(B) \in 5\#15$. Ihre Ausführungszeiten seien jeweils 8. Diese Konstellation ist in Abbildung 5.1 gezeigt. Hier wie in den folgenden Abbildungen von Aufgaben ist eine Aufgabe $t$ durch ein Rechteck (Zeitfenster) gekennzeichnet, dessen linker Rand $est(t)$ und dessen rechter Rand $lct(t)$ darstellt. Die Ausführungszeit $dur(t)$ ist durch ein schraffiertes Rechteck innerhalb des erlaubten Zeitfensters für $t$ verdeutlicht. Das Ende dieses Rechtecks bezeichnet $ect(t)$. Schiebt man das schraffierte Rechteck im Zeitfenster ganz nach rechts, so markiert dessen linker Rand nun $lst(t)$.

**Abbildung 5.1** Reifikation für disjunktive Schedulingprobleme

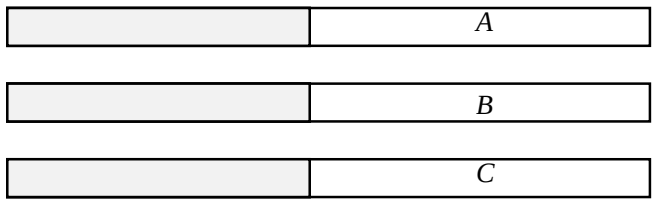



Da im Beispiel $ect(A) > lst(B)$ gilt, gilt $N_p(d) = \bot$ für den Projektor $p$, der $compl(A) \leq start(B)$ vollständig realisiert ($A$ kann nicht vor $B$ fertig werden, da sonst für $B$ nicht mehr genügend Zeit verbliebe). Der Projektor für den anderen reifizierten Constraint zeigt nun das Propagierungsverhalten für den Constraint $compl(B) \leq start(A)$. Somit wird der Bereich von $start(A)$ zu 13#15 und der von $start(B)$ zu 5#7 eingeschränkt.

Ein Nachteil dieser Realisierung durch Reifikation ist die große Zahl von Projektoren pro Ressource $r$, nämlich $3 \cdot (util(r)^2 - util(r))/2$ viele. Dieser Nachteil kann umgangen werden, indem man einen einzelnen Projektor verwendet, der für eine Ressource $r$ genau das gleiche Propagierungsverhalten hat wie alle Projektoren für die reifizierten Constraints für die Ressource $r$ zusammen. Neben der Verringerung des Speicherverbrauchs ergibt sich in einer konkreten Implementierung auch ein besseres Laufzeitverhalten, da für die gleiche Propagierung nur ein Projektor ausgeführt werden muß und nicht $O(util(r)^2)$ viele. In Kapitel 10 wird ein Problem vorgestellt, für das ein solcher Projektor sehr gute Ergebnisse liefert.

Das folgende Beispiel zeigt aber, daß die Propagierung, die durch die Realisierung durch reifizierte Constraints erreicht wird, recht schwach ist (siehe Abbildung 5.2). Seien drei Aufgaben $A, B, C$ auf einer Ressource gegeben, sei der Bereichsconstraint $start(A) \in 1\#10 \wedge start(B) \in 1\#10 \wedge start(C) \in 1\#10$ und sei die Ausführungszeit jeweils 8. Offensichtlich können die Aufgaben nicht überlappungsfrei plaziert werden. Realisieren wir für die Paare $(A,B)$, $(A,C)$ und $(B,C)$ jedoch die reifizierten Constraints, so kann keine Propagierung stattfinden, da für kein Paar $(x,y)$ die Ungleichung $ect(x) > lst(y)$ oder umgekehrt gilt. Betrachten wir aber das insgesamt zur Verfügung stehende Zeitintervall zwischen 1 und 18 (der spätesten Fertigstellungszeit für $A$, $B$ und $C$), und die insgesamt benötigte Ausführungszeit von $3 \cdot 8 = 24$, so wird sofort klar, daß es keine Serialisierung für die drei Aufgaben geben kann. Eine solche globalere Sicht auf die Aufgaben, die die gleiche Ressource benötigen, ist der Kern einer stärkeren Propagierungstechnik. Diese Technik berücksichtigt alle Variablen gleichzeitig, für die ein Kapazitätsconstraint gilt, und ist unentbehrlich zur Lösung einiger harter Standardprobleme aus dem Operations Research (siehe Abschnitt 9.4).

**Abbildung 5.2** Reifikation ist nicht ausreichend für disjunktive Schedulingprobleme




### 5.2.2 Edge-Finding

Die prinzipiellen Ideen für ein besseres Verfahren zur Realisierung von Kapazitätsconstraints stammen aus dem Operations Research [CP89]. Der hierfür verwendete Begriff *Edge-Finding* wurde in [AC91] geprägt (siehe auch Abschnitt 5.4). In diesem Abschnitt übertragen wir die Ideen von Edge-Finding auf das Konzept von Propagierung und Distribuierung. Insbesondere prüfen wir die Eignung der aufgestellten Propagierungsregeln für Propagierungsfunktionen von Projektoren. Wir führen zuerst einige Schreibweisen ein.

**Definition 17** *Im folgenden bezeichnet S eine (evtl. leere) Menge von Aufgaben, die alle auf der gleichen Ressource ausgeführt werden. Wir verallgemeinern die Schreibweisen, die für Aufgaben eingeführt wurden, wie folgt.*

| | |
|---|---|
| $dur(S) = \sum_{t \in S} dur(t)$ | *$dur(S)$ ist die Ausführungszeit von S* |
| $res(S) = r$ | *$res(S)$ ist die Ressource von S, falls $res(t) = r$ für alle $t \in S$* |
| $est(S) = \min(\{est(t) \mid t \in S\})$ | *$est(S)$ ist die frühest mögliche Startzeit einer Aufgabe in S* |
| $lct(S) = \max(\{lct(t) \mid t \in S\})$ | *$lct(S)$ ist die spätest mögliche Fertigstellungszeit einer Aufgabe in S* |

Die Argumentation am Ende des vorangehenden Abschnitts läßt sich nun verallgemeinern.

**Proposition 9** *Gilt bzgl. eines Bereichsconstraints $d_1$*

$$lct(S) - est(S) < dur(S),$$

*so kann es für das zugehörige Schedulingproblem keine Lösung geben. Dies gilt auch bzgl. aller Bereichsconstraints $d_2$, für die $d_2 \models d_1$ gilt.*

Beweis. Die Korrektheit der ersten Behauptung ist offensichtlich. Die zweite Behauptung gilt, da $dur(S)$ konstant ist und $lct(S)$ bzgl. eines stärkeren Bereichsconstraints nur kleiner und $est(S)$ nur größer werden kann. □

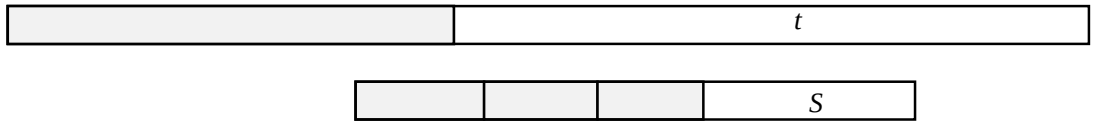
Sei jetzt eine Aufgabe $t$ und eine Menge von Aufgaben $S$ mit $res(S) = res(t)$ gegeben, so daß $t \notin S$ gilt. Die folgenden Ungleichungen werden für Tests benutzt, ob $t$ vor oder nach allen Aufgaben von $S$ plaziert werden kann oder ob $t$ zwischen zwei Aufgaben von $S$ plaziert werden kann (daher auch der Begriff Edge-Finding, das heißt, daß Begrenzungen von Aufgabenmengen gefunden werden). Die Schlüsse, die aus diesen Tests gezogen werden können, erlauben es, die möglichen Start- und Fertigstellungszeiten von Aufgaben zu reduzieren.

Gilt die Ungleichung

$$lct(S) - est(S) < dur(S) + dur(t), \tag{5.3}$$

so reicht der Platz zwischen $est(S)$ und $lct(S)$ nicht aus, um $S$ und $t$ zu plazieren. Enthält $S$ mindestens zwei Aufgaben, entspricht dies der Tatsache, daß $t$ nicht zwischen zwei Aufgaben in $S$ plaziert werden kann, da die dafür notwendige Zeitdauer $dur(S) + dur(t)$ nicht zur Verfügung steht. Ein Beispiel ist in Abbildung 5.3 gezeigt. Hierbei wird die Menge $S$ durch ein gemeinsames Zeitintervall von (hier) drei Aufgaben dargestellt. Beachte, daß die Reihenfolge der Aufgaben in $S$ noch nicht feststehen muß.

**Abbildung 5.3** Aufgabe $t$ kann nicht zwischen zwei Aufgaben in $S$ plaziert werden

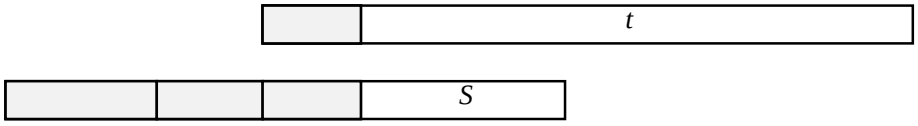



Gilt die Ungleichung

$$lct(S) - est(t) < dur(S) + dur(t), \tag{5.4}$$

so kann $t$ nicht vor allen Aufgaben in $S$ ausgeführt werden. Würde $t$ nämlich vor $S$ ausgeführt, so stünde nur das Zeitintervall $lct(S) - est(t)$ zur Verfügung, was aber nicht ausreicht, um sowohl $S$ als auch $t$ zu plazieren. Ein Beispiel ist in Abbildung 5.4 gezeigt.

**Abbildung 5.4** Aufgabe $t$ kann nicht vor allen Aufgaben in $S$ ausgeführt werden

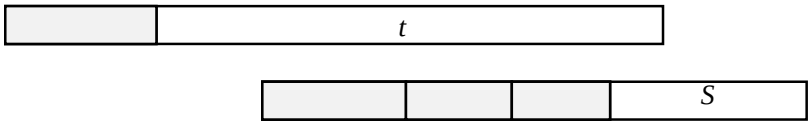



Gilt die Ungleichung

$$lct(t) - est(S) < dur(S) + dur(t), \tag{5.5}$$

so kann $t$ nicht nach allen Aufgaben in $S$ ausgeführt werden. Würde $t$ nämlich nach $S$ ausgeführt, so stünde nur das Zeitintervall $lct(t) - est(S)$ zur Verfügung. Abbildung 5.5 zeigt ein Beispiel.

**Abbildung 5.5** Aufgabe $t$ kann nicht nach allen Aufgaben in $S$ ausgeführt werden




Wir werden nun eine Reihe von Propagierungsregeln vorstellen, die Grundlage für die Realisierung von Projektoren für Kapazitätsconstraints sein werden. Dazu nehmen wir einen Bereichsconstraint $d$ in einer Konfiguration $(d,P)$ an, die eine Spezifikation $(d,C)$ für ein Schedulingproblem realisiert. Wir sagen dann, daß für die Reduktion $(d,P) \rightarrow (d',P)$ der Bereich einer Variablen $start(t)$ durch $start(t) \geq m$ reduziert wird, wenn $d' \wedge c(P) \models\!\models d \wedge c(P)$ gilt und $d'$ sich nur im Bereich von $start(t)$ von $d$ unterscheidet, nämlich

$$dom(start(t),d') = dom(start(t),d) \cap m\#\max(dom(start(t),d))$$

für $m \leq \max(dom(start(t),d))$. Entsprechend gilt für $start(t) \leq m$ dann

$$dom(start(t),d') = \min(dom(start(t),d))\#m \cap dom(start(t),d)$$

für $m \geq \min(dom(start(t),d))$. Analog für $compl(t) \geq m$ bzw. $compl(t) \leq m$. Diese Reduktionen können dann die Basis für Propagierungsfunktionen von Projektoren bilden. Wir nehmen hier an, daß diese Reduktionen zu keinem leeren Bereich führen (später werden diese Fälle für eine zu realisierende Propagierungsfunktion natürlich berücksichtigt). Wir setzen für die Beweise der folgenden Propositionen voraus, daß in der betrachteten Konfiguration Kapazitätsconstraints für alle Ressourcen realisiert sind.

**Proposition 10** *Gilt Ungleichung (5.4), so muß vor $t$ mindestens eine Aufgabe aus $S$ plaziert werden. Damit kann der Bereich von $start(t)$ durch*

$$start(t) \geq \min(\{ect(t') | t' \in S\})$$

*reduziert werden.*

Beweis. Die rechte Seite der Ungleichung bezeichnet den frühesten Zeitpunkt, zu dem eine Aufgabe in $S$ fertiggestellt werden kann. Damit folgt die Behauptung direkt aus (5.4). □

**Proposition 11** *Gilt Ungleichung (5.5), so muß nach t mindestens eine Aufgabe aus S plaziert werden, und der Bereich von $compl(t)$ kann durch*

$$compl(t) \leq \max(\{lst(t') | t' \in S\})$$

*reduziert werden.*

Beweis. Der Wert $c = \max(\{lst(t') | t' \in S\})$ bezeichnet den spätesten Zeitpunkt, zu dem eine Aufgabe in $S$ beginnen kann. Damit folgt aus Ungleichung (5.5), daß $t$ spätestens zum Zeitpunkt $c$ fertiggestellt sein muß. □

**Proposition 12** *Gelten gleichzeitig Ungleichung (5.3) und Ungleichung (5.4), so muß t nach allen Aufgaben in S plaziert werden. Damit kann der Bereich von $start(t)$ durch*

$$start(t) \geq \max(\{est(S) + dur(S), \max(\{ect(t') | t' \in S\})\})$$

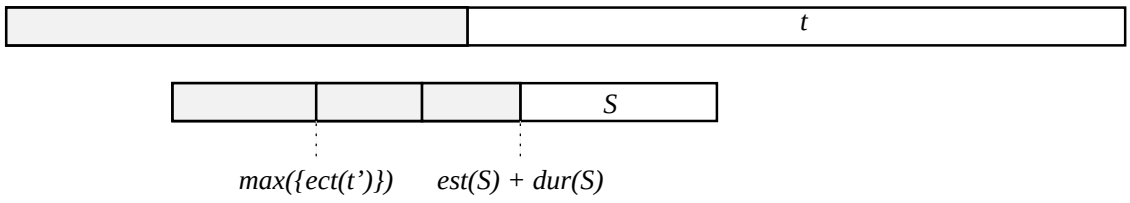
*reduziert werden. Umgekehrt, muß jede Aufgabe in S vor t fertiggestellt sein:*

$$\forall t' \in S : compl(t') \leq lst(t).$$

Beweis. Die erste Ungleichung folgt aus (5.3) und (5.4) sowie der Tatsache, daß sowohl $est(S) + dur(S)$ als auch $\max(\{ect(t') | t' \in S\})$ eine untere Schranke für die Fertigstellung aller Aufgaben in $S$ ist. $lst(t)$ ist eine obere Schranke für die Startzeit von $t$. Damit muß jedes $t' \in S$ spätestens zum Zeitpunkt $lst(t)$ fertiggestellt sein. □

Ein Beispiel für die Anwendbarkeit von Proposition 12 ist in Abbildung 5.6 gezeigt.

**Abbildung 5.6** Aufgabe $t$ muß nach allen Aufgaben in $S$ plaziert werden




**Proposition 13** *Gelten gleichzeitig Ungleichung (5.3) und Ungleichung (5.5), so muß t vor allen Aufgaben in S plaziert werden. Der Bereich von $compl(t)$ kann somit durch*

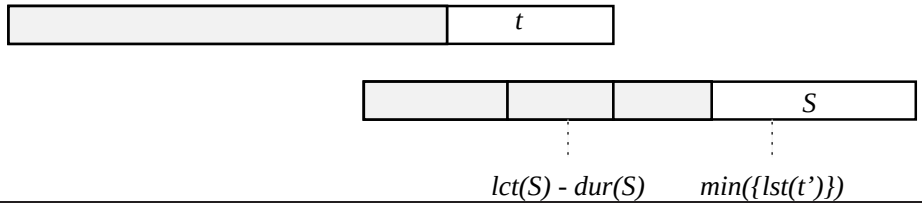
$$compl(t) \leq \min(\{lct(S) - dur(S), \min(\{lst(t') | t' \in S\})\})$$

*reduziert werden. Umgekehrt, muß jede Aufgabe in S nach t beginnen:*

$$\forall t' \in S : start(t') \geq ect(t).$$

Beweis. $c_1 = lct(S) - dur(S)$ bzw. $c_2 = \min(\{lst(t') | t' \in S\})$ ist eine obere Schranke für die späteste Startzeit von $S$, so daß $S$ fertiggestellt werden kann. Damit folgt aus (5.3) und (5.5), daß

**Abbildung 5.7** Aufgabe *t* muß vor allen Aufgaben in *S* plaziert werden




$t$ spätestens zum Zeitpunkt $\min(\{c_1, c_2\})$ fertiggestellt sein muß. Da $ect(t)$ eine untere Schranke für die Fertigstellungszeit von $t$ ist, gelten die übrigen Ungleichungen. □

Ein Beispiel für die Anwendbarkeit von Proposition 12 ist in Abbildung 5.7 gezeigt.

Es kann jetzt noch der Fall betrachtet werden, daß $t \in S$ gilt. In diesem Fall kann man analog zu den Ungleichungen (5.3) bis (5.5) die folgenden Ungleichungen überprüfen. Sei dazu $S' = S \setminus \{t\}$.

$$lct(S') - est(S') < dur(S) \tag{5.6}$$

$$lct(S') - est(t) < dur(S) \tag{5.7}$$

$$lct(t) - est(S') < dur(S) \tag{5.8}$$

Gilt (5.6), so kann $t$ nicht zwischen zwei Aufgaben von $S'$ plaziert werden. Gilt (5.7), so kann $t$ nicht vor allen Aufgaben in $S'$ plaziert werden. Gilt (5.8), so kann $t$ nicht nach allen Aufgaben in $S'$ plaziert werden. Entsprechend können wieder Bereiche reduziert werden.

Die Regeln mit Edge-Finding können besonders viele Bereiche reduzieren, wenn für eine Menge von Aufgaben $S$ der Wert von $lct(S) - est(S) - dur(S)$ relativ klein ist, die Ressource also besonders stark beansprucht ist. Betrachtet man dann noch eine Aufgabe $t$ mit vergleichsweise großer Ausführungszeit, so ist es wahrscheinlich, daß eine Regel angewendet werden kann. Siehe auch Abschnitt 9.4.1 für eine genauere Analyse anhand von konkreten Beispielen.

Um die Regeln in den vorangehenden Propositionen für Projektoren zu verwenden, muß noch ihre Monotonie gezeigt werden. Das heißt, jede Regel kann für einen Bereichsconstraint mindestens genauso stark Bereiche von Variablen reduzieren wie für einen schwächeren Bereichsconstraint. Dazu definiere jede Proposition eine Funktion $N : \mathcal{D} \to \mathcal{D}$, die einen Bereichsconstraint $d$ auf einen Bereichsconstraint $d'$ abbildet (ähnlich wie die Propagierungsfunktion $N_p$ für einen Projektor $p$).

**Proposition 14** *Die Regeln in den Propositionen 10 bis 13 zur Reduktion von Bereichen sind monoton. Das heißt, gilt für zwei Bereichsconstraints $d_2 \models d_1$, so gilt auch $N(d_2) \models N(d_1)$ für die jeweiligen Funktionen $N$, die von den Propositionen definiert werden.*

Beweis. Es soll exemplarisch Proposition 13 bewiesen werden. Gilt Ungleichung (5.3) bzgl. $d_1$, so auch bzgl. $d_2$, da $lct(S)$ nur kleiner und $est(S)$ nur größer werden kann und $dur(S) + dur(t)$ konstant ist. Entsprechend gilt auch Ungleichung (5.5) bzgl. $d_2$, da $lct(t)$ nur kleiner werden kann. Somit sind die Bedingungen der Regeln erfüllt und sie können auch in $d_2$ angewendet werden. Da aber auch $ect(t)$ nur größer und $lst(t')$ nur kleiner werden kann, gilt $N(d_2) \models N(d_1)$. □

Somit könnten Proposition 9 bis 13 direkt dazu benutzt werden, eine Propagierungsfunktion $N_p$ eines Projektors $p$ für einen Kapazitätsconstraint zu implementieren. Neben Korrektheit und Monotonie können auch die weiteren Eigenschaften von Projektoren in Abschnitt 2.4 gewährleistet werden. Es ergibt sich aber das Problem, daß pro Ressource $r$ es $2^{util(r)}$ mögliche Aufgabenmengen $S$ gibt und man bei einem naiven Vorgehen $util(r) \cdot 2^{util(r)}$ Paare $(t,S)$ betrachten müßte, um die obigen Schlüsse ziehen zu können. Dies ist für Anwendungen nicht realistisch.

In [CP90] wurde zum ersten Mal gezeigt, wie die gesamten Schlüsse aus den Propagierungsregeln der Propositionen 9, 12 und 13 in einem Algorithmus gezogen werden können, dessen Laufzeit quadratisch in der Zahl der Aufgaben wächst. Die zugrundeliegende Technik verwendet aber recht komplizierte Hilfskonstruktionen (siehe auch Abschnitt 5.4).

Eine Alternative ist es, nur eine Teilmenge der möglichen Aufgabenmengen zu betrachten. Es ist aber nicht klar, welche feste Teilmenge dies für ein Problem sein soll, damit die Regeln auch oft angewendet werden können. Deshalb ist es vorteilhaft, die Teilmengen dynamisch zu erzeugen, um vielversprechende Kandidaten zu finden. Die Mengen werden dabei jeweils durch die aktuellen Bereichsgrenzen bestimmt. Diese Idee liegt den meisten gegenwärtig erfolgreichen Implementierungen für constraintbasiertes Lösen von Schedulingproblemen zugrunde [BPN95a, CL94a, Wür96].

Beachte, daß bei Edge-Finding für disjunktive Schedulingprobleme nur Werte an den Grenzen der Variablenbereiche entfernt werden. Dies ist wichtig für die Kombination von Projektoren für Kapazitätsconstraints mit Projektoren für weitere Constraints. Wählt man für diese weiteren Constraints nämlich Projektoren aus, die sehr stark propagieren, indem sie zum Beispiel Lücken in Bereichen erzeugen, so werden die Projektoren für Kapazitätsconstraints diese Lücken nicht ausnutzen können. Wir werden aber in Abschnitt 5.3 exemplarisch auch Kapazitätsconstraints vorstellen, die Lücken in Bereichen erzeugen können.

### 5.2.3 Aufgabenintervalle

In diesem Abschnitt stellen wir einen konkreten Algorithmus mit Edge-Finding vor, der zur Definition einer Propagierungsfunktion herangezogen werden kann. Dazu verwenden wir sogenannte *Aufgabenintervalle* (engl. *task intervals*), die eine bestimmte Menge von Aufgabenmengen definieren (siehe [CL94a] und [CL95]). Wir übertragen einen Algorithmus aus [CL94a], der in einer regel-basierten Sprache formuliert ist, in eine Propagierungsfunktion für Projektoren.

**Definition 18** *Seien Aufgaben $t_1$ und $t_2$ gegeben mit $res(t_1) = res(t_2), est(t_1) \leq est(t_2), lct(t_1) \leq lct(t_2)$. Dann ist das Aufgabenintervall $I(t_1,t_2)$ die Menge*

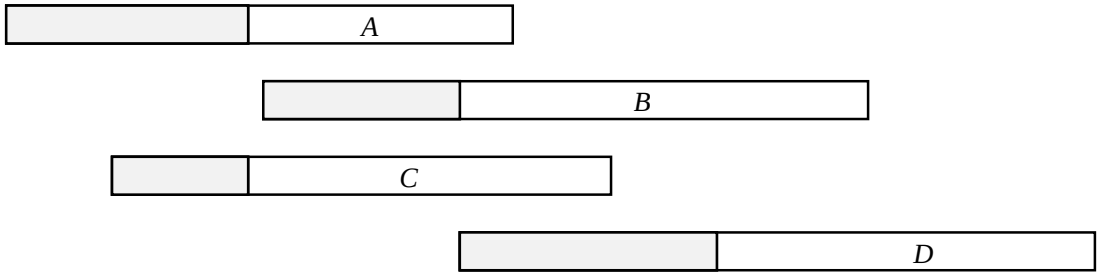
$$I(t_1,t_2) = \{t \in T | res(t) = res(t_1), est(t_1) \leq est(t), lct(t) \leq lct(t_2)\}.$$

Abbildung 5.8 zeigt ein Beispiel. Das Aufgabenintervall $I(A,B)$ enthält die Aufgaben $A$, $B$ und $C$ und das Aufgabenintervall $I(C,D)$ enthält $B$, $C$ und $D$.

Für eine Ressource $r$ gibt es $util(r)^2$ (nicht notwendigerweise verschiedene) Aufgabenintervalle.

Algorithmus 5.1 definiert eine mögliche Propagierungsfunktion $N_p$ für einen Projektor $p$, der einen Kapazitätsconstraint durch Propagierung mit Edge-Finding auf Aufgabenintervallen realisiert. Dabei nehmen wir an, daß $N_p$ auf einen Bereichsconstraint $d$ einer Konfiguration $(d,P)$

**Abbildung 5.8** Aufgabenintervalle




angewendet wird. Die Reduktion von Bereichen mit $start(t) \leq m$ etc. liefert einen neuen Bereichsconstraint gemäß den Propositionen im vorangehenden Abschnitt. Dieser neue Bereichsconstraint wird dann im Algorithmus dazu verwendet, weitere Reduktionen durchzuführen. Der Bereichsconstraint $d'$, der als letztes von dem Algorithmus berechnet worden ist, ist das Ergebnis von $N_p(d)$. Es ist dann also die Reduktion $(d,P) \rightarrow (d',P)$ möglich. Falls in diesem Algorithmus eine Bereichsreduzierung durch $start(t) \leq m$ usw. zu einem leeren Bereich führen würde, wird der Algorithmus automatisch abgebrochen und liefert $\perp$ als Ergebnis zurück. Wir nehmen als Subsumtionsfunktion die triviale Funktion an, die Subsumtion erst für einen determinierten Bereichsconstraint erkennt. Für die folgenden Algorithmen in diesem Kapitel gelten die gleichen Annahmen.

Der Algorithmus bestimmt für eine Ressource $r$ alle möglichen Aufgabenintervalle darauf. Anschließend werden für jedes dieser Intervalle $S$ die Tests aus dem vorangehenden Abschnitt mit jeweils allen Aufgaben $t$ auf $r$ durchgeführt. Dabei wird solange iteriert, bis keine Propagierung mehr möglich ist. Somit wird Idempotenz garantiert.

Sei $k$ die maximale Kardinalität eines Bereichs von $start(t)$ und $compl(t)$ für alle $t \in T$ mit $res(t) = r$. Wir haben nun zu zeigen, daß Algorithmus 5.1 keine der für Propagierungsfunktionen für Projektoren notwendigen Eigenschaften verletzt. Dann könnte der Algorithmus zur Implementierung eines Projektors herangezogen werden. Definiere dazu Algorithmus 5.1 eine Propagierungsfunktion $N$ auf Bereichsconstraints. Wir sagen, daß $N$ monoton ist, falls aus $d_2 \models d_1$ auch $N(d_2) \models N(d_1)$ folgt.

**Satz 4** *Für die durch Algorithmus 5.1 definierte Propagierungsfunktion gelten alle Eigenschaften aus Definition 4 außer Monotonie. Der Algorithmus hat eine Laufzeitkomplexität von* $\Omega(util(r)^3)$ *und* $O(k \cdot util(r)^4)$.

Beweis. Wir betrachten zuerst die Laufzeitkomplexität. Wir nehmen zur Bestimmung der unteren Schranke an, daß die Berechnung nicht $\perp$ als Ergebnis hat (siehe auch Kapitel 4). Die untere Schranke der Komplexität ergibt sich aus den drei ineinander geschachtelten Schleifen. Die obere Schranke ergibt sich aus der Tatsache, daß bis zum Erreichen des Fixpunktes der Propagierung (bis kein Bereich mehr reduziert werden kann) höchstens $O(k \cdot util(r))$ Werte aus den Bereichen entfernt werden können. Beachte, daß bis zum Erreichen des Fixpunktes gerechnet werden muß, da für die Propagierungsfunktionen Idempotenz gefordert wird.

Alle Eigenschaften aus Definition 4 außer Monotonie ergeben sich direkt aus den Propositionen, die als Grundlage der Propagierung herangezogen werden. Bei der Frage nach der Mo-

**Algorithmus 5.1** Edge-Finding mit Aufgabenintervallen

```
repeat
  for t1 ∈ T, res(t1) = r do
    for t2 ∈ T, res(t2) = r do
      if est(t1) ≤ est(t2) and
         lct(t1) ≤ lct(t2) then
        S := {t ∈ T | res(t) = r, est(t1) ≤ est(t), lct(t) ≤ lct(t2)};
        if lct(S) − est(S) < dur(S) then
          // Proposition 9 gilt
          return ⊥;
        end;
        for t ∈ T, res(t) = r do
          if t ∉ S then
            if lct(S) − est(S) ≥ dur(S) + dur(t) then
              if lct(S) − est(t) < dur(S) + dur(t) then
                // Proposition 10 gilt
                start(t) ≥ min({ect(t') | t' ∈ S}); end;
              if lct(t) − est(S) < dur(S) + dur(t) then
                // Proposition 11 gilt
                compl(t) ≤ max({lst(t') | t' ∈ S}); end;
            else
              if lct(S) − est(t) < dur(S) + dur(t) then
                // Proposition 12 gilt
                start(t) ≥ max({est(S) + dur(S), max({ect(t') | t' ∈ S})});
                ∀t' ∈ S : compl(t') ≤ lst(t);
              else if lct(t) − est(S) < dur(S) + dur(t) then
                // Proposition 13 gilt
                compl(t) ≤ min({lct(S) − dur(S), min({lst(t') | t' ∈ S})});
                ∀t' ∈ S : start(t') ≥ ect(t);
              end;
            end;
          else   // t in S
            ...
          end;
        end;
      end;
    end;
  end;
until kein Bereich kann mehr reduziert werden
```

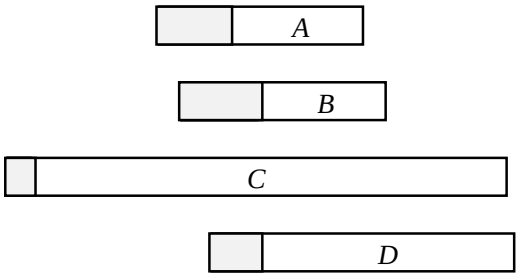
notonie müssen wir beachten, daß bei jedem Aufruf des Algorithmus die Aufgabenintervalle in Abhängigkeit von den aktuellen Bereichsgrenzen neu berechnet werden. Dies bedeutet aber insbesondere, daß die Aufgabenintervalle nach einer Verstärkung des Bereichsconstraints nicht unbedingt die gleichen Mengen denotieren müssen wie zuvor.

Der Beweis, daß Monotonie nicht gilt, beruht auf einem Gegenbeispiel in [Bap94]. Seien vier Aufgaben mit folgenden Bereichen der zugehörigen Startzeitvariablen und Ausführungszeiten gegeben.

| Aufgabe $t$ | $start(t)$ | $dur(t)$ | $lct(t)$ |
|---|---|---|---|
| $A$ | 6#11 | 3 | 14 |
| $B$ | 7#12 | 3 | 15 |
| $C$ | 0#19 | 1 | 20 |
| $D$ | 8#18 | 2 | 20 |

Wir betrachten das Aufgabenintervall $S = I(A,B) = \{A,B\}$ (siehe Abbildung 5.9).

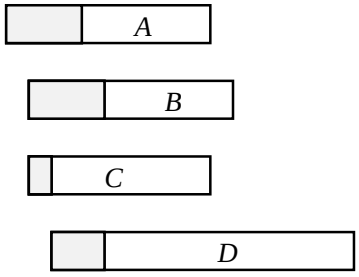
**Abbildung 5.9** Die untere Schranke von $D$ kann erhöht werden




Da $lct(B) - est(D) = 7 < dur(S) + dur(D) = 8$ gilt, kann $D$ nicht vor allen Aufgaben in $S$ plaziert werden. Da aber auch $lct(B) - est(A) = 9 \geq dur(S) + dur(D)$ gilt, kann $D$ möglicherweise zwischen $A$ und $B$ plaziert werden. Die Startzeit von $D$ kann somit wegen Proposition 10 zu $\min(\{ect(A), ect(B)\}) = 9$ angehoben werden.

Um zu einem Gegenbeispiel zu kommen, betrachten wir den Fall, daß der Bereich von $start(C)$ sich zu 7#13 geändert hat (siehe Abbildung 5.10).

**Abbildung 5.10** Die untere Schranke von $D$ kann nicht erhöht werden




Das Aufgabenintervall $S = I(A,B)$ umfaßt nun die Aufgaben $A$, $B$ und $C$. Es gilt wieder $lct(B) - est(D) = 7 < dur(S) + dur(D) = 9$. Auch gilt $lct(B) - est(A) = 9 \geq dur(S) + dur(D)$.

Jedoch beträgt der Wert von $\min(\{ect(A), ect(B), ect(C)\})$ nur $ect(C) = 8$. Damit wird die untere Bereichsgrenze von $start(D)$ nicht mehr angehoben. Da auch durch andere Aufgabenintervalle der Bereich von $start(D)$ nicht entsprechend reduziert wird, ist die durch Algorithmus 5.1 definierte Propagierungsfunktion somit nicht monoton.

Während in [Bap94] die Nicht-Monotonie nur für den Fall $t \notin S$ gezeigt wird, kann auch ein Gegenbeispiel für den Fall $t \in S$ konstruiert werden. Hierzu nimmt man den Bereichsconstraint $start(A) \in 7\#12 \wedge start(B) \in 7\#13 \wedge start(C) \in 0\#19 \wedge start(D) \in 9\#14$ an und betrachtet wieder das Intervall $I(A,B)$. Aufgrund der Propagierungsregeln kann die untere Schranke von $start(D)$ auf 10 erhöht werden. Wenn jetzt der Bereich von $start(C)$ auf 7#14 geändert wird, kann man wie oben nur noch $start(D) \geq 8$ folgern. □

Beachte, daß in den üblichen Komplexitätsanalysen [CL94a, BPN95a, Bap94] nicht berücksichtigt wird, daß für einen Projektor Idempotenz gelten sollte.

Wegen der fehlenden Monotonie-Eigenschaft könnte Algorithmus 5.1 also nicht für einen Projektor verwendet werden, da Konfluenz der Berechnung nicht mehr gewährleistet werden kann. Wir werden in den Kapiteln 7 und 11 aber sehen, daß mit entsprechenden Vorkehrungen in der Implementierung auch Algorithmus 5.1 für einen Projektor herangezogen werden kann (da die Projektoren für Kapazitätsconstraints erst nach der Ausführung aller anderen Projektoren ausgeführt werden und bei jeder Berechnung genau die gleiche Reihenfolge eingehalten wird).

Der nachfolgende Satz zeigt jedoch, daß eine monotone Propagierungsfunktion sehr wohl mit Aufgabenintervallen konstruiert werden kann, die aber etwas schwächer propagiert. Dies ist nach unserem Wissen das erste Mal, daß Monotonie für eine Propagierungsfunktion mit Aufgabenintervallen nachgewiesen wird. In DFKI Oz verwenden wir aber den stärker propagierenden Algorithmus 5.1.

**Satz 5** *Verwendet man in Algorithmus 5.1 nur die Propagierungsregeln aus Proposition 9, 12 und 13, so ist die dadurch definierte Propagierungsfunktion auch monoton.*

gaga

Beweis. Der Beweis soll nur für die Regeln aus Proposition 12 geführt werden. Für Proposition 13 ergibt sich ein analoger Beweis. Der Beweis für Proposition 9 ist offensichtlich.

Wir haben zu zeigen, daß für die so definierte Propagierungsfunktion $N$ und zwei Bereichsconstraints $d_1$ und $d_2$ mit $d_2 \models d_1$ auch $N(d_2) \models N(d_1)$ gilt.

Gelten bzgl. eines Bereichsconstraints $d_1$ die Ungleichungen (5.3) und (5.4) für ein Aufgabenintervall $S = I(t_1,t_2)$ und eine Aufgabe $t \notin S$. Sei $low = est(t_1)$ und $up = lct(t_2)$ bzgl. $d_1$. Gilt $d_2 \models d_1$, so gilt $dom(start(t), d_2) \subseteq dom(start(t), d_1)$.

Sei $t'_1 \in S$, so daß $est(t'_1) = \min(dom(start(t'_1), d_2)) \geq low$ gilt und es kein $t'' \in S$ gibt, so daß $est(t'') < est(t'_1)$ bzgl. $d_2$ gilt. Sei $t'_2 \in S$, so daß $lct(t'_2) = \max(dom(compl(t''), d_2)) \leq up$ gilt und es kein $t'' \in S$ gibt, so daß $lct(t'') > lct(t'_1)$ bzgl. $d_2$ gilt. Bezeichne $S'$ das Aufgabenintervall $I(t'_1,t'_2)\backslash\{t\}$. Wegen der Konstruktion des Algorithmus wird auch dieses Aufgabenintervall betrachtet werden. Beachte, daß $t$ nicht in $S'$ enthalten ist.

Offensichtlich gilt $dur(S') \geq dur(S)$, da höchstens noch Aufgaben zusätzlich zu $S'$ hinzugekommen sein können (neben den Aufgaben in $S$). Aus $d_2 \models d_1$ folgt $\min(dom(start(t), d_1)) \leq$

$\min(dom(start(t), d_2))$. Es gilt auch $lct(S') \leq up$ und $est(S') \geq low$ bzgl. $d_2$. Damit sind aber die Ungleichungen (5.3) und (5.4) für $S'$ und $t$ in $d_2$ erfüllt und die entsprechenden Bereichsreduktionen können angewendet werden.

Wir müssen jetzt noch zeigen, daß der aus $S'$ und $t$ resultierende Bereich bzgl. $d_2$ kleiner ist als derjenige, der aus $S$ und $t$ bzgl. $d_1$ resultiert. Dazu betrachten wir zuerst den Bereich von $start(t)$. Der Wert von $est(S') + dur(S')$ kann bzgl. $d_2$ nicht kleiner sein als $est(S) + dur(S)$ bzgl. $d_1$. Sei nun $v$ der Wert von $\max(\{ect(t')|t' \in S\})$ bzgl. $d_1$. Sind in $S'$ die gleichen Aufgaben wie in $S$ enthalten, so kann $v' = \max(\{ect(t')|t' \in S'\})$ bzgl. $d_2$ nicht kleiner als $v$ sein. Falls eine neue Aufgabe $t'$ in $S'$ hinzukommt, die nicht in $S$ enthalten war, so kann jedoch $v'$ auch nicht kleiner sein als $v$. Damit ist der Bereich für $start(t)$ bzgl. $d_2$ nicht größer als derjenige bzgl. $d_1$. Für alle $t' \in S'$ ergibt sich das gleiche Resultat für $compl(t')$, da in diesem Fall $lst(t)$ bzgl. $d_2$ nicht größer sein kann als bzgl. $d_1$. Somit ist die Behauptung bewiesen. □

In diesem Abschnitt haben wir also gezeigt, wie Aufgabenintervalle zur Implementierung eines Projektors eingesetzt werden können, der einen Kapazitätsconstraint realisiert.

Es stellt sich aber die Frage, ob man durch Verwendung von Aufgabenintervallen gegenüber den exponentiell vielen Aufgabenmengen wirklich Propagierung verliert. Dies ist zwar richtig, doch decken Aufgabenintervalle sehr viele der 'interessanten' Aufgabenmengen ab. Zudem können wir zeigen, daß alle Schlüsse aus beliebigen Kombinationen von Aufgabenmengen und Aufgaben für die Propositionen 9, 12 und 13 auch mit Aufgabenintervallen gezogen werden können. Sei dazu $S$ eine beliebige Aufgabenmenge. Ist $S$ ein Aufgabenintervall, so gilt die Behauptung trivialerweise. Ist $S$ kein Aufgabenintervall, so gibt es eine Aufgabe $t$ mit $est(S) \leq est(t)$ und $lct(t) \leq lct(S)$, aber $t \notin S$. Sei $S'$ dasjenige Aufgabenintervall mit $est(S) = est(S')$ und $lct(S) = lct(S')$, so daß $t \in S'$ gilt. Offensichtlich (wegen $dur(S') \geq dur(S)$) können die erwähnten Propositionen angewendet werden und der erhaltene Bereichsconstraint ist stärker als derjenige, der durch $S$ erhalten wird. Für die Propositionen 10 und 11 gilt dies jedoch nicht mehr. Ein Gegenbeispiel ist in Abbildung 5.10 enthalten. Betrachtet man die Aufgabenmenge $\{A, B\}$, so kann ein stärkerer Bereichsconstraint abgeleitet werden, als mit dem Aufgabenintervall $I(A, B) = \{A, B, C\}$.

Im folgenden Abschnitt werden wir einen Algorithmus vorstellen der nur eine Laufzeitkomplexität von $\Omega(util(r)^2)$ besitzt. Daß solche Algorithmen in der Praxis ausreichend sind, wird in den Fallstudien deutlich.

### 5.2.4 Der Zwei-Phasen-Algorithmus

In diesem Abschnitt stellen wir für einen Kapazitätsconstraint eine Propagierungsfunktion für einen Projektor vor, deren Laufzeit im besten Fall quadratisch in der Zahl der Aufgaben pro Ressource wächst. Die benutzte Technik (genauer die Auswahl der betrachteten Aufgabenmengen zur Propagierung) geht auf [MS96] zurück, in dem sie als ein Teil eines umfangreicheren Algorithmus zur Bestimmung von unteren Schranken für Schedulelängen verwendet wird. Wir werden hier diesen Algorithmus für Constraintprogrammierung verwenden. Außerdem werden wir den in [MS96] vorgestellten Algorithmus um weitere Propagierungsregeln verstärken und seine Eigenschaften (insbesondere Monotonie) überprüfen.

Der hier vorgestellte Algorithmus, der der Propagierungsfunktion zugrundeliegt, besteht aus zwei

Phasen. In jeder Phase wird eine bestimmte Menge von Aufgabenmengen konstruiert und Propagierung mit Edge-Finding durchgeführt. Dies wird so lange wiederholt, bis keine Reduktion von Bereichen mehr möglich ist. Wir nennen diese zwei Phasen *Aufwärts-* und *Abwärtsphase* aufgrund der Art wie die Aufgabenmengen konstruiert und bei der Propagierung berücksichtigt werden. In jeder Phase wird nur ein Teil der Propositionen zum Edge-Finding berücksichtigt. So finden in der Aufwärtsphase nur Proposition 9 und 10 sowie eine Spezialisierung von Proposition 12 Anwendung. Hingegen werden in der Abwärtsphase Proposition 9 und 11 sowie eine Spezialisierung von Proposition 13 verwendet.

Die quadratische Laufzeit des Algorithmus beruht auf der speziellen Art, wie die Aufgabenmengen für das Edge-Finding konstruiert werden und mit welchen Aufgaben die Tests aus Abschnitt 5.2.2 durchgeführt werden. Es werden zwar für eine gegebene Ressource $r$ auch hier Aufgabenintervalle betrachtet, aber anstatt $util(r)^2$ werden hier höchstens $util(r)$ Intervalle erzeugt. Da dieser Algorithmus in der Regel weniger stark propagiert als der Algorithmus im vorangehenden Abschnitt, muß man für Anwendungen zwischen Laufzeiteffizienz und Propagierungsstärke abwägen. Ein quantitativer Vergleich ist in Abschnitt 9.4 zu finden.

Wir beschreiben zuerst die sogenannte *Aufwärtsphase*. Dazu wird für jede Ressource $r$ und für alle Aufgaben $k$ mit $res(k) = r$ eine Reihe von Aufgabenmengen konstruiert und Propagierung mit Edge-Finding durchgeführt.

Sei $r \in R$ eine Ressource und $k$ eine Aufgabe mit $res(k) = r$. Dann wird die Menge $S^0$ der Aufgaben, die zwischen $est(k)$ und $lct(k)$ plaziert werden können, wie folgt berechnet.

$$S^0 = \{t \in T | res(t) = r, est(k) \leq est(t), lct(t) \leq lct(k)\}$$

Sei $O$ die Menge von Aufgaben $\{t \in T | res(t) = r, t \notin S^0\}$. Die Funktion `maxEst` extrahiert aus einer Aufgabenmenge diejenige Aufgabe $t$ mit dem größten Wert von $est(t)$. Dann werden Aufgabenmengen $S^i$ wie folgt konstruiert.

```
i := 1;
while O ≠ ∅ do
  t := maxEst(O);
  O := O\{t};
  if est(t) < est(k) and lct(t) ≤ lct(k) then
     S^i := S^(i-1) ∪ {t};
     i := i+1;
  end;
end;
```

Seien $S^0 \subset \cdots \subset S^b$ die so konstruierten Mengen. Beachte, daß aus der Konstruktion der Mengen $S^i$ die Ungleichungen

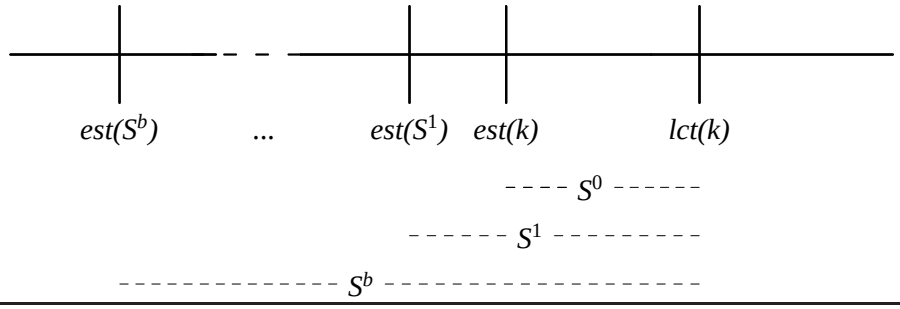
$$est(S^0) > est(S^1) \geq \cdots \geq est(S^b)$$

folgen (siehe auch Abbildung 5.11). Gilt für ein $S^i$ die Ungleichung

$$lct(S^i) - est(S^i) < dur(S^i),$$

so kann keine Lösung des zugehörigen Schedulingproblems (bzw. des zugehörigen Schedulingproblems der Spezifikation, die von der betrachteten Konfiguration realisiert wird) existieren und die zu definierende Propagierungsfunktion wird $\perp$ als Ergebnis liefern.

**Abbildung 5.11** Die Aufgabenmengen $S^i$ der Aufwärtsphase




Aufgrund der besonderen Konstruktion der Aufgabenmengen $S^i$ können wir nun die folgende Proposition zur Reduktion von Bereichen angeben, die Proposition 12 spezialisiert.

**Proposition 15** *Gelten für eine Aufgabenmenge $S^i$ und eine Aufgabe t der Aufwärtsphase gleichzeitig Ungleichung (5.3) und Ungleichung (5.4), so muß t nach allen Aufgaben in $S^i$ plaziert werden. Damit kann der Bereich von $start(t)$ durch*

$$start(t) \geq \max(\{est(S^j) + dur(S^j) | 0 \leq j \leq i\} \cup \{ect(t') | t' \in S^i\})$$

*reduziert werden. Umgekehrt, muß jede Aufgabe in $S^i$ vor t fertiggestellt sein:*

$$\forall t' \in S^i : compl(t') \leq lst(t).$$

Beweis. Der Beweis wird analog zu dem Beweis von Proposition 12 geführt, wobei die spezielle Konstruktion der Mengen $S^i$ berücksichtigt wird. □

Sei $L$ die Menge der Aufgaben, deren spätest mögliche Fertigstellungszeit größer als die spätest mögliche Fertigstellungszeit der Aufgabe $k$ ist:

$$L = \{t \in T | res(t) = r, lct(t) > lct(k)\}$$

Die Aufwärtsphase wird abgeschlossen durch die Ausführung von Algorithmus 5.2, der die Bereiche von Variablen gemäß den Propagierungsregeln in Abschnitt 5.2.2 reduziert. Da wir von $S^b$ startend zu immer größeren Werten von $est(S^i)$ übergehen, heißt diese Phase Aufwärtsphase. Wie schon Algorithmus 5.1 wird auch dieser Algorithmus automatisch abgebrochen und liefert $\perp$ als Ergebnis zurück, falls eine Bereichsreduzierung durch $start(t) \geq m$ usw. zu einem leeren Bereich führen würde. Die Funktion `maxDur` extrahiert aus einer Aufgabenmenge die Aufgabe mit der größten Ausführungszeit. Die Tests für die Propagierungsregeln werden zwischen den Mengen $S^i$ und den Aufgaben in $L$ durchgeführt. Durch die Konstruktion der Mengen $S^i$ und der Wahl der Elemente aus $L$ läßt sich die Laufzeit begrenzen. Es werden nämlich nur $b + |L|$ Tests durchgeführt.

Durch die Betrachtung von lediglich $b + |L|$ Kombinationen in Algorithmus 5.2, anstatt der insgesamt $b \cdot |L|$ möglichen, geht aber weniger Propagierung verloren als es vielleicht zuerst den Anschein hat. Beachte, daß in Algorithmus 5.2 die Aufgabenmenge $S^i$ von der weiteren Betrachtung ausgeschlossen wird, wenn $t$ zwischen $est(S^i)$ und $lct(S^i)$ plaziert werden kann. Wegen der

**Algorithmus 5.2** Die Propagierung der Aufwärtsphase

```
i := b;
while i ≥ 0 and L ≠ ∅ do
  t := maxDur(L);
  if lct(S^i) − est(S^i) ≥ dur(S^i) + dur(t) then
    if lct(S^i) − est(t) < dur(S^i) + dur(t) then
      // Proposition 10 gilt
      start(t) ≥ min({ect(t')|t' ∈ S^i});
    else L = L\{t};
    end;
    i := i-1;
  else
    if lct(S^i) − est(t) < dur(S^i) + dur(t) then
      // Proposition 15 gilt
      start(t) ≥ max({est(S^j) + dur(S^j)|0 ≤ j ≤ i} ∪ {ect(t')|t' ∈ S^i});
      ∀t' ∈ S^i : compl(t') ≤ lst(t);
    end;
    L := L\{t};
  end;
end;
```

Wahl des Elements $t$ aus $L$ gilt diese Tatsache nämlich auch für alle anderen Elemente in $L$ (die eine kleinere Ausführungszeit als $t$ haben) und Proposition 15 kann für kein $l \in L$ gelten. Gilt zusätzlich, daß $t$ vor allen Aufgaben von $S^i$ plaziert werden kann, wird $t$ aus $L$ entfernt. Wegen der Konstruktion der $S^i$ gilt diese Tatsache nämlich auch für alle $S^j$ mit $j < i$ und Proposition 10 kann für kein $S^j$ und dieses $t$ gelten. Kann $t$ nicht zwischen zwei Aufgaben von $S^i$ plaziert werden, so wird $t$ aus $L$ entfernt. Kann nämlich dann $t$ vor allen Aufgaben von $S^i$ plaziert werden, so gilt dies auch für alle $S^j$ mit $j < i$ und Proposition 15 kann für kein $S^j$ und dieses $t$ gelten. Kann $t$ nicht vor allen Aufgaben von $S^i$ plaziert werden, so kann kein $S^j$ mit $j < i$ eine stärkere Reduzierung des Bereichs von $t$ bewirken als $S^i$.

Nachdem die Aufwärtsphase beendet wurde, beginnt die *Abwärtsphase*. Dazu wird wieder für jede Ressource $r$ und für alle Aufgaben $k$ mit $res(k) = r$ eine Reihe von Aufgabenmengen konstruiert und Propagierung mit Edge-Finding durchgeführt.

Sei $r$ eine Ressource und $k$ eine Aufgabe auf $r$. Dann wird wieder die Menge $S^0$ der Aufgaben, die zwischen $est(k)$ und $lct(k)$ plaziert werden können, berechnet.

$$S^0 = \{t \in T | res(t) = r, est(k) \leq est(t), lct(t) \leq lct(k)\}$$

Sei $O$ wieder die Menge von Aufgaben $\{t \in T | res(t) = r, t \notin S^0\}$. Die Funktion `minLct` extrahiert aus einer Aufgabenmenge diejenige Aufgabe $t$ mit dem kleinsten Wert von $lct(t)$. Dann werden die Aufgabenmengen $S^i$ wie folgt konstruiert.

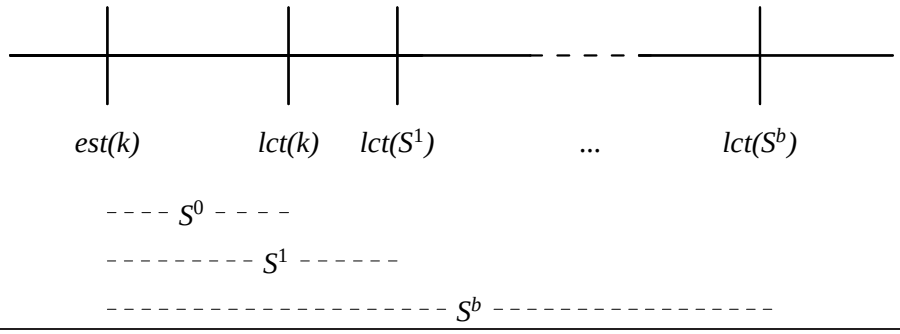
```
i := 1;
while O ≠ ∅ do
  t := minLct(O);
  O := O\{t};
  if est(t) ≥ est(k) and lct(t) > lct(k) then
     S^i = S^(i−1) ∪ {t};
```

```
      i := i+1;
  end;
end;
```

Seien $S^0 \subset \cdots \subset S^b$ die konstruierten Mengen. Beachte, daß aus der Konstruktion der Mengen $S^i$ die Ungleichungen

$$lct(S^0) < lct(S^1) \leq \cdots \leq lct(S^b)$$

**Abbildung 5.12** Die Aufgabenmengen $S^i$ der Abwärtsphase




folgen (siehe Abbildung 5.12). Gilt für ein $S^i$ die Ungleichung

$$lct(S^i) - est(S^i) < dur(S^i),$$

so kann wie in der Aufwärtsphase keine Lösung des zugehörigen Schedulingproblems existieren und die zu definierende Propagierungsfunktion wird $\perp$ als Ergebnis liefern.

Die folgende Proposition zur Reduktion von Bereichen spezialisiert Proposition 13.

**Proposition 16** *Gelten für eine Aufgabenmenge $S^i$ und eine Aufgabe $t$ der Abwärtsphase gleichzeitig Ungleichung (5.3) und Ungleichung (5.5), so muß $t$ vor allen Aufgaben in $S^i$ plaziert werden. Der Bereich von $compl(t)$ kann somit durch*

$$compl(t) \leq \min(\{lct(S^j) - dur(S^j) | 0 \leq j \leq i\} \cup \{lst(t') | t' \in S^i\})$$

*reduziert werden. Umgekehrt, muß jede Aufgabe in $S^i$ nach $t$ beginnen:*

$$\forall t' \in S^i : start(t') \geq ect(t).$$

Beweis. Der Beweis wird analog zu dem Beweis von Proposition 13 geführt, wobei die spezielle Konstruktion der Mengen $S^i$ berücksichtigt wird. □

Sei $L$ die Menge der Aufgaben, deren frühest mögliche Startzeit kleiner als die frühest mögliche Startzeit der Aufgabe $k$ ist:

$$L = \{t \in T | res(t) = r, est(t) < est(k)\}$$

Die Abwärtsphase wird abgeschlossen durch Ausführung von Algorithmus 5.3.

Wie die Aufwärtsphase wird die Abwärtsphase für alle Aufgaben auf einer Ressource ausgeführt.

**Algorithmus 5.3** Die Propagierung der Abwärtsphase

```
i := b;
while i ≥ 0 and L ≠ ∅ do
  t := maxDur(L);
  if lct(S^i) − est(S^i) ≥ dur(S^i) + dur(t) then
    if lct(t) − est(S^i) < dur(S^i) + dur(t) then
      // Proposition 11 gilt
      compl(t) ≤ max({lst(t') | t' ∈ S^i});
    else L := L\{t};
    end;
    i := i-1;
  else
    if lct(t) − est(S^i) < dur(S^i) + dur(t) then
      // Proposition 16 gilt
      compl(t) ≤ min({lct(S^j) − dur(S^j) | 0 ≤ j ≤ i} ∪ {lst(t') | t' ∈ S^i});
      ∀t' ∈ S^i : start(t') ≥ ect(t);
    end;
    L := L\{t};
  end;
end;
```

Im Gegensatz zu dem hier vorgestellten Zwei-Phasen-Algorithmus wird in [MS96] nur Propagierung verwendet, die ähnlich den Propositionen 15 und 16 ist.

Der Gesamtalgorithmus (*Zwei-Phasen-Algorithmus* genannt) besteht aus der Hintereinanderausführung der Aufwärts- und Abwärtsphase bis kein Bereich einer Variablen mehr reduziert werden kann. Wir haben nun zu prüfen, ob dieser Zwei-Phasen-Algorithmus eine Propagierungsfunktion für einen Projektor zur Realisierung eines Kapazitätsconstraints definieren kann.

Sei $k$ die maximale Kardinalität eines Bereichs von $start(t)$ und $compl(t)$ für alle $t \in T$ mit $res(t) = r$.

**Satz 6** *Für die durch den Zwei-Phasen-Algorithmus definierte Propagierungsfunktion gelten alle Eigenschaften aus Definition 4 außer Monotonie. Der Algorithmus hat eine Laufzeitkomplexität von* $\Omega(util(r)^2)$ *bzw.* $O(k \cdot util(r)^3)$.

Beweis. Wir betrachten zuerst die Laufzeitkomplexität. In einem Schritt des Gesamtalgorithmus werden jeweils $util(r)$ Auf- und Abwärtsphasen hintereinander durchgeführt. In einer $j$-ten Phase werden $b_j$ Aufgabenmengen $S_j^i$ konstruiert. Dies ist inklusive der Berechnung der Werte $dur(S_j^i)$ etc. in $O(util(r))$ möglich. Die konstruierte Menge $L_j$ hat $util(r) - |S_j^0| - b_j$ Elemente. Da die jeweiligen Schleifen somit höchstens $util(r)$ mal durchlaufen werden, ergibt sich die untere Schranke für die Laufzeitkomplexität. Beachte, daß die Aufgaben bereits vor Beginn der zwei Phasen jeweils nach fallender Ausführungszeit, nach fallender frühest möglicher Startzeit und steigender spätest möglicher Fertigstellungszeit sortiert werden können. Die obere Schranke ergibt sich aus der Anzahl der höchstens zu entfernenden Werte in den Bereichen und der Tatsache, daß bis zum Erreichen eines Fixpunktes der Bereichsreduktion gerechnet wird.

Alle Eigenschaften aus Definition 4 außer Monotonie ergeben sich direkt aus den Propositionen, die als Grundlage der Propagierung herangezogen werden.

Daß die so definierte Propagierungsfunktion nicht monoton ist, ergibt sich mit dem gleichen Gegenbeispiel wie für die Aufgabenintervalle in Abschnitt 5.2.3, da der Zwei-Phasen-Algorithmus in diesem Fall die gleichen Aufgabenmengen konstruiert. □

Wie Algorithmus 5.1 kann jedoch auch der Zwei-Phasen-Algorithmus für einen Projektor verwendet werden, wenn die Implementierung Konfluenz sicherstellen kann (siehe auch Seite 71).

**Satz 7** *Verwendet man im Zwei-Phasen-Algorithmus nur die Propagierungsregeln aus Proposition 9, 15 und 16, so ist die dadurch definierte Propagierungsfunktion auch monoton.*

Beweis. Der Beweis soll nur für die Regeln aus Proposition 15 (für die Aufwärtsphase) geführt werden. Für Proposition 16 ergibt sich der Beweis analog. Für Proposition 9 ist der Beweis offensichtlich.

Wir haben zu zeigen, daß für die so definierte Propagierungsfunktion $N$ und zwei Bereichsconstraints $d_1$ und $d_2$ mit $d_2 \models d_1$ auch $N(d_2) \models N(d_1)$ gilt.

Gelten bzgl. eines Bereichsconstraints $d_1$ die Ungleichungen (5.3) und (5.4) für eine Aufgabenmenge $S^i$, eine Aufgabe $k \in S^i$ mit $lct(k) = lct(S^i)$ und eine Aufgabe $t$ mit $lct(t) > lct(k)$. Sei $low = est(S^i)$ und $up = lct(S^i)$ bzgl. $d_1$. Gilt $d_2 \models d_1$, so gilt $dom(start(t), d_2) \subseteq dom(start(t), d_1)$.

Wir betrachten jetzt die Bereiche der Aufgaben bzgl. $d_2$. Sei $t_1 \in S^i$, so daß $est(t_1) \geq low$ gilt und es kein $t' \in S^i$ gibt, so daß $est(t') < est(t_1)$ gilt. Sei $t_2 \in S^i$, so daß $lct(t_2) \leq up$ gilt und es kein $t' \in S^i$ gibt, so daß $lct(t') > lct(t_1)$ gilt. Sei $S^{i'}$ das Aufgabenintervall $I(t_1, t_2)$ mit $est(S^{i'}) = est(t_1)$ und $lct(S^{i'}) = lct(t_2)$. Aufgrund seiner Konstruktion wird der Zwei-Phasen-Algorithmus bzgl. $d_2$ diese Menge $S^{i'}$ betrachten. Es gilt $dur(S^{i'}) \geq dur(S^i)$, da höchstens noch Aufgaben zusätzlich zu $S^{i'}$ hinzugekommen sein können (neben den Aufgaben in $S^i$).

Nun nehmen wir eine Fallunterscheidung vor. Gilt $t \in S^{i'}$ bzgl. $d_2$, so kann sofort gefolgert werden, daß kein Schedule existiert (da für jede konstruierte Aufgabenmenge die Ungleichung aus Proposition 9 geprüft wird und (5.3) für $S^i$ und $t$ bereits bzgl. $d_1$ galt).

Gilt $t \notin S^{i'}$ bzgl. $d_2$, so können wir zwei Fälle unterscheiden. Gilt $lct(t) > lct(S^{i'})$ bzgl. $d_2$, so sind offensichtlich auch die Ungleichungen (5.3) und (5.4) für $S^{i'}$ und $t$ erfüllt und die daraus folgenden Bereichsreduzierungen können angewendet werden. Mit der gleichen Argumentation wie im Beweis von Satz 5 sind die resultierenden Bereiche für $S^{i'}$ und $t$ bzgl. $d_2$ nicht größer als diejenigen, die aus $S^i$ und $t$ bzgl. $d_1$ resultieren.

Gilt $est(t) < est(S^{i'})$ bzgl. $d_2$ und $lct(t) \leq lct(S^{i'})$ bzgl. $d_2$, so definiert $t$ eine Aufgabenmenge $S^j$ mit $est(S^j) = est(t)$ und $lct(S^j) = lct(S^{i'})$ bzgl. $d_2$. $S^j$ enthält mindestens die Aufgaben aus $S^{i'}$, womit $dur(S^j) \geq dur(S^{i'})$ gilt. Weiterhin wird auch für $S^j$ die Ungleichung aus Proposition 9 geprüft. Da aber Ungleichung (5.4) bzgl. $d_1$ galt, gilt die Ungleichung $lct(S^j) - est(S^j) < dur(S^j) + dur(t)$ bzgl. $d_2$, und es kann kein Schedule existieren.

Damit ist die Behauptung bewiesen. □

## 5.3 Kumulative Schedulingprobleme

In diesem Abschnitt betrachten wir Schedulingprobleme, bei denen auf einer Ressource mehr als eine Aufgabe gleichzeitig ausgeführt werden kann ($cap(r) > 1$).

**Definition 19** *Eine Ressource r heißt* kumulativ, *falls* $cap(r) > 1$ *gilt. Ein Schedulingproblem heißt* kumulativ, *falls mindestens eine Ressource dieses Problems kumulativ ist.*

Der Begriff kumulatives Scheduling wurde wohl zuerst in [AB93] geprägt, worin jedoch keine Algorithmen vorgestellt wurden. In [AB93] können für Aufgaben jedoch auch variable Ausführungszeiten und Ressourcenverbrauch gegeben sein. Im Operations Research ist für diese Problemklasse auch der Begriff Resource-Constrained Project Scheduling (RCSP) gebräuchlich (siehe Abschnitt 5.4). Wir verwenden hier jedoch den Begriff des kumulativen Schedulings, der in der Constraintprogrammierung üblich ist [AB93, CL96b, ILOG96a]. Wir werden zeigen, daß einige der Propagierungsregeln für disjunktive Schedulingprobleme aus Abschnitt 5.2.2 verallgemeinert werden können (Abschnitt 5.3.1). Mit diesen Regeln formulieren wir einen verallgemeinerten Zwei-Phasen-Algorithmus und weisen dessen Monotonie nach. Jedoch ist die erzielte Propagierung nicht so stark wie in Abschnitt 5.2.2 für disjunktive Probleme beschrieben, da sich die Ausführung von Aufgaben im kumulativen Fall zeitlich überlappen darf und deshalb die Propagierungsregeln abgeschwächt werden müssen. Somit wird ein ergänzendes Verfahren vorgestellt, das für kumulative Probleme zu mehr Propagierung führt (Abschnitt 5.3.2).

Zuerst betrachten wir ein Beispiel. Dazu seien drei Aufgaben *A*, *B* und *C* auf einer Ressource mit einer Kapazität 8 gegeben. Die möglichen Startzeiten (die Bereiche), die Ausführungszeit und der Ressourcenverbrauch sind in der folgenden Tabelle angegeben.

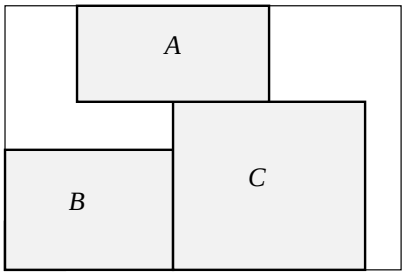
| Aufgabe $t$ | $start(t)$ | $dur(t)$ | $use(t)$ |
|---|---|---|---|
| A | 6#9 | 6 | 3 |
| B | 4#12 | 5 | 4 |
| C | 4#11 | 6 | 5 |

Da die Ausführung der Aufgaben sich zeitlich überlappen darf, ist die Darstellung der möglichen Plazierungen nicht mehr so einfach wie für disjunktive Schedulingprobleme. Wir verwenden die Konvention, daß nur die Ausführungsrechtecke mit der Größe $size(t)$ dargestellt werden. Für die möglichen frühesten Start- und spätesten Fertigstellungszeiten sind die entsprechend angegebenen Bereiche zu konsultieren. Ein mit dünnen Linien eingezeichnetes Rechteck bezeichnet die insgesamt zur Verfügung stehende Zeitdauer (gemäß der *lct* der Aufgaben) und Kapazität der betrachteten Ressource. Das vorgestellte Beispiel kann dann wie in Abbildung 5.13 dargestellt werden. Ein konkreter Schedule könnte jedoch sehr wohl *C* vor *B* ausführen und *A* weiter ‘nach rechts’ verschieben (also später starten lassen als die in Abbildung 5.13 eingezeichnete Position).

### 5.3.1 Eine Verallgemeinerung von Edge-Finding

Zuerst zeigen wir, wie die in Abschnitt 5.2.2 betrachteten Propagierungsregeln für disjunktive Schedulingprobleme für kumulative Probleme verallgemeinert werden können. Diese neuen

**Abbildung 5.13** Beispiel für ein kumulatives Schedulingproblem




Propagierungsregeln werden dann in eine generalisierte Form des Zwei-Phasen-Algorithmus aus Abschnitt 5.2.4 integriert.

Sei $S$ wieder eine Menge von Aufgaben, die auf der gleichen Ressource $r$ plaziert werden sollen, und sei $t$ eine Aufgabe mit $res(t) = r$ und $t \notin S$. Sei weiterhin $size(S) = \sum_{t \in S} size(t)$.

**Proposition 17** *Gilt bzgl. eines Bereichsconstraints $d_1$ die Ungleichung*

$$(lct(S) - est(S)) \cdot cap(r) < size(S), \tag{5.9}$$

*so kann es für das zugehörige Schedulingproblem keine Lösung geben. Dies gilt auch bzgl. aller Bereichsconstraints $d_2$, für die $d_2 \models d_1$ gilt.*

Beweis. Die Korrektheit der ersten Behauptung ist offensichtlich. Die zweite Behauptung gilt, da $size(S)$ und $cap(S)$ konstant sind und $lct(S)$ bzgl. eines stärkeren Bereichsconstraints nur kleiner und $est(S)$ nur größer werden kann. □

Es können jetzt wieder Propagierungsregeln formuliert werden, die die Bereiche der Variablen für Start- und Fertigstellungszeiten reduzieren. Dabei verallgemeinern wir Proposition 12 und 13 aus Abschnitt 5.2.2. Für die Propositionen 10 und 11 gelingt dies nicht, da im kumulativen Fall die Ausführung von Aufgaben sich zeitlich überlappen darf. Beachte, daß wir selbst für die Verallgemeinerung der Propositionen 12 und 13 keine so starke Propagierung erhalten wie im disjunktiven Fall.

Wir schreiben $\lceil c \rceil$ für die kleinste ganze Zahl, die größer oder gleich $c$ ist.

**Proposition 18** *Gelte die Ungleichung*

$$(lct(S) - est(S)) \cdot cap(r) < size(S) + size(t) \tag{5.10}$$

*und die Ungleichung*

$$(lct(S) - est(t)) \cdot cap(r) < size(S) + size(t). \tag{5.11}$$

*Sei weiterhin*

$$\delta = size(S) - (lct(S) - est(S)) \cdot (cap(r) - use(t)).$$

*Gilt $\delta > 0$, so kann der Bereich von $start(t)$ durch*

$$start(t) \geq est(S) + \lceil \delta / use(t) \rceil$$

*reduziert werden.*

Beweis. Der Beweis ist an einen Beweis in [Nui94] angelehnt. Gelten die Ungleichungen (5.10) und (5.11), so folgt, daß es in $S$ keine Aufgabe gibt, die später als der Fertigstellungszeitpunkt $compl(t)$ beginnt. Gäbe es in $S$ nämlich eine solche Aufgabe, so würde $compl(t) < lct(S)$ gelten. Die benötigte Größe zur Plazierung von $S \cup \{t\}$ zwischen $\min(\{est(S), est(t)\})$ und $lct(S)$ wäre also dann mindestens $size(S) + size(t)$. Da die beiden Ungleichungen (5.10) und (5.11) aber äquivalent zu

$$(lct(S) - \min(\{est(S), est(t)\})) \cdot cap(r) < size(S) + size(t) \tag{5.12}$$

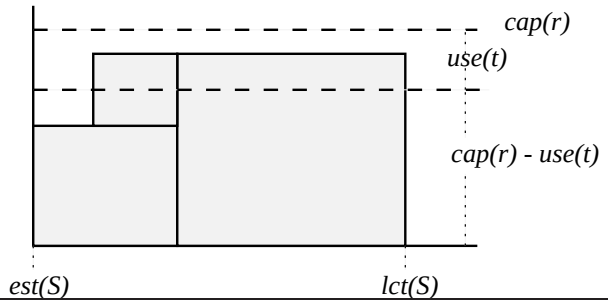
sind, kann es in $S$ keine solche Aufgabe gegen.

Nun ist eine untere Schranke für die Startzeit von $t$ zu finden (siehe auch Abbildung 5.14). Ohne einen Startwert von $t$ zu verbieten, können Aufgaben aus $S$ plaziert werden, die insgesamt eine Größe $d = (lct(S) - est(S)) \cdot (cap(r) - use(t))$ haben. Gilt $d \geq size(S)$, so kann der Bereich von $start(t)$ nicht weiter reduziert werden, da $d$ ausreicht, um $S$ zu plazieren. Gilt $d < size(S)$, so muß noch eine Aufgabengröße von $size(S) - d$ plaziert werden. Da aber $t$ den Verbrauch $use(t)$ hat, muß diese Menge komplett vor dem Beginn von $t$ plaziert werden ($t$ 'blockiert' den Streifen der Höhe $use(t)$ und keine Aufgabe in $S$ darf nach $compl(t)$ beginnen). Eine untere Schranke für die Fertigstellung dieser Menge ist $est(S) + \lceil (size(S) - d)/use(t) \rceil$. Damit ergibt sich die Behauptung. □

Wendet man diese Proposition auf disjunktive Schedulingprobleme an, so erhält man lediglich die Reduzierung $start(t) \geq est(S) + dur(S)$ (vgl. Proposition 12).

Beachte, daß die Propagierungsregel $start(t) \geq \lceil size(S)/cap(r) \rceil$ nicht korrekt wäre, da $S$ zum Beispiel nur aus einer einzigen Aufgabe bestehen könnte, für die $use(S) + use(t) \leq cap(r)$ und $\lceil size(S)/cap(r) \rceil > 0$ gilt. Die Aufgabe $t$ kann aber sehr wohl zeitgleich zu der in $S$ enthaltenen Aufgabe ausgeführt werden.

**Abbildung 5.14** Beispiel für Proposition 18




**Proposition 19** *Gelte die Ungleichung*

$$(lct(S) - est(S)) \cdot cap(r) < size(S) + size(t) \tag{5.13}$$

*und die Ungleichung*

$$(lct(t) - est(S)) \cdot cap(r) < size(S) + size(t). \tag{5.14}$$

*Sei weiterhin*

$$\delta = size(S) - (lct(S) - est(S)) \cdot (cap(r) - use(t)).$$

*Gilt* $\delta > 0$*, so kann der Bereich von* $compl(t)$ *durch*

$$compl(t) \leq lct(S) - \lceil \delta / use(t) \rceil$$

*reduziert werden*

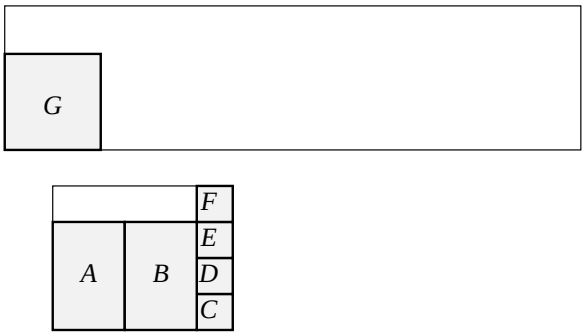
Beweis. Der Beweis wird analog dem Beweis von Proposition 18 geführt. □

Entsprechende Propositionen können auch für den Fall $t \in S$ formuliert werden.

Nun soll als Beispiel eine Aufgabe $G$ und die Aufgabenmenge $S$ betrachtet werden, die aus den Aufgaben $A, B, C, D, E, F$ besteht. Die Bereiche der zugehörigen Startzeitvariablen, Ausführungszeit und Verbrauch aller Aufgaben sind in der folgenden Tabelle beschrieben. $cap(r)$ sei 4. Das Beispiel ist in Abbildung 5.15 gezeigt.

| Aufgabe $t$ | $start(t)$ | $dur(t)$ | $use(t)$ | $lct(t)$ |
|---|---|---|---|---|
| A | 5#8 | 2 | 3 | 10 |
| B | 5#8 | 2 | 3 | 10 |
| C | 5#9 | 1 | 1 | 10 |
| D | 5#9 | 1 | 1 | 10 |
| E | 5#9 | 1 | 1 | 10 |
| F | 5#9 | 1 | 1 | 10 |
| G | 4#15 | 3 | 3 | 18 |

**Abbildung 5.15** Beispiel für kumulatives Scheduling




Da $(lct(S) - est(S)) \cdot cap(r) = (10-5) \cdot 4 = 20$ kleiner als $size(S) + size(G) = 16+9 = 25$ ist, kann $G$ nicht vollständig im Intervall $est(S)$ bis $lct(S)$ plaziert werden.

Da auch $(lct(S) - est(G)) \cdot 4 = (10-4) \cdot 4 = 24$ kleiner als 25 ist, kann somit insgesamt Proposition 18 angewendet werden.

Ohne eine Einschränkung für $G$ läßt sich nur die Größe $(lct(S) - est(S)) \cdot (cap(r) - use(G)) = 5$ plazieren. Der Rest $size(S) - 5 = 16 - 5 = 11$ muß möglichst früh plaziert werden. Damit ergibt sich eine untere Schranke für die Startzeit von $G$ von $est(S) + \lceil 11/use(G) \rceil = 5+4 = 9$. Dies entspricht dem Schedule, daß die Aufgaben mit Verbrauch Eins gleichzeitig zu den Aufgaben mit Ausführungszeit Zwei ausgeführt werden.

Wie für disjunktive Probleme, kann auch für den kumulativen Fall ein Zwei-Phasen-Algorithmus entwickelt werden. Wir werden hier nur die Aufwärtsphase betrachten. Die Abwärtsphase läßt sich analog umsetzen.

Sei $r \in R$ eine Ressource und $k$ eine Aufgabe mit $res(k) = r$. Dann wird die Menge $S^0$ der Aufgaben, die zwischen $est(k)$ und $lct(k)$ plaziert werden können, wie folgt berechnet.

$$S^0 = \{t \in T | res(t) = r, est(k) \leq est(t), lct(t) \leq lct(k)\}$$

Sei $O$ die Menge von Aufgaben $\{t \in T | res(t) = r, t \notin S^0\}$. Die Aufgabenmengen $S^i$ werden wie folgt konstruiert.

```
i := 1;
while O ≠ ∅ do
  t := maxEst(O);
  O := O\{t};
  if est(t) < est(k) and lct(t) ≤ lct(k) then
     S^i := S^(i-1) ∪ {t};
     i := i+1;
  end;
end;
```

Seien $S^0 \subset \cdots \subset S^b$ die so konstruierten Mengen. Beachte, daß aus der Konstruktion der Mengen $S^i$ die Ungleichungen

$$est(S^0) > est(S^1) \geq \cdots \geq est(S^b)$$

folgen. Gilt für ein $S^i$ die Ungleichung

$$(lct(S^i) - est(S^i)) \cdot cap(r) < size(S^i),$$

so kann keine Lösung des zugehörigen Schedulingproblems existieren.

Sei $L$ die Menge der Aufgaben, deren spätest mögliche Fertigstellungszeit größer als die spätest mögliche Fertigstellungszeit der Aufgabe $k$ ist:

$$L = \{t \in T | res(t) = r, lct(t) > lct(k)\}$$

Die Aufwärtsphase wird abgeschlossen durch die Ausführung von Algorithmus 5.4.

Sei $k$ die maximale Kardinalität eines Bereichs von $start(t)$ und $compl(t)$ für alle $t \in T$ mit $res(t) = r$.

**Satz 8** *Für die durch den Zwei-Phasen-Algorithmus für kumulatives Scheduling definierte Propagierungsfunktion gelten alle Eigenschaften aus Definition 4. Der Algorithmus hat eine Laufzeitkomplexität von* $\Omega(util(r)^2)$ *bzw.* $O(k \cdot util(r)^3)$.

Beweis. Die Beweise erfolgen analog den Beweisen der Sätze 6 und 7 für disjunktive Probleme. □

Beachte, daß die so definierte Propagierungsfunktion insbesondere auch monoton ist. Dies liegt daran, daß zur Propagierung nur die Verallgemeinerungen der Proposition 12 und 13 verwendet werden.

**Algorithmus 5.4** Die Aufwärtsphase für kumulatives Scheduling

```
i := b;
while i ≥ 0 and L ≠ ∅ do
  S := S^i;
  t := maxDur(L);
  if (lct(S^i) − est(S^i)) · cap(r) ≥ size(S^i) + size(t) then
    if (lct(S^i) − est(t)) · cap(r) ≥ size(S^i) + size(t) then
      L := L\{t}; end;
    i := i-1;
  else
    if (lct(S^i) − est(t)) · cap(r) < size(S^i) + size(t) then
      // Proposition 18 gilt
      δ := size(S^i) − (lct(S^i) − est(S^i)) · (cap(r) − use(t));
      if δ > 0 then
        start(t) ≥ est(S^i) + ⌈δ/use(t)⌉; end;
    end;
    L := L\{t};
  end;
end;
```

### 5.3.2 Histogramm-Propagierung

In diesem Abschnitt werden Propagierungsregeln für kumulative Schedulingprobleme vorgestellt, die den verallgemeinerten Zwei-Phasen-Algorithmus ergänzen. Die Idee beruht darauf, Intervalle zu bestimmen, in denen eine Aufgabe garantiert ihre Ressource verbrauchen wird. Diese Intervalle können dann zur Definition von geeigneten Propagierungsregeln verwendet werden.

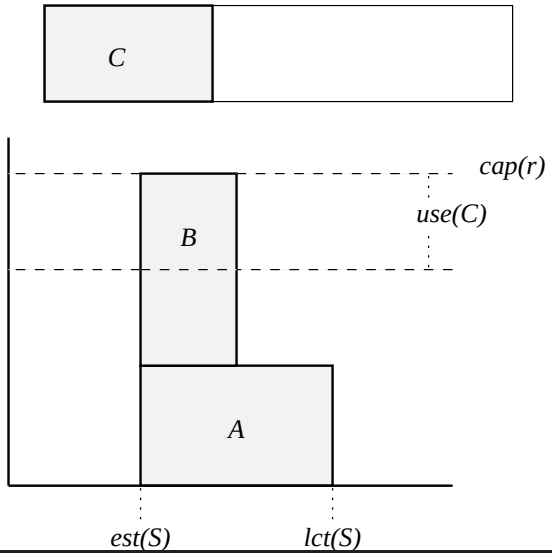
Die grundlegenden Techniken gehen auf [ELT91] zurück. Auch dort wird der garantierte Ressourcenverbrauch in bestimmten Intervallen berechnet und daraus Propagierung abgeleitet. Während der hier vorgestellte Algorithmus jedoch eine quadratische Laufzeitkomplexität in der Anzahl der Aufgaben je Ressource besitzt, läuft der Algorithmus in [ELT91] in kubischer Zeitkomplexität. Dies ist darauf zurückzuführen, daß in [ELT91] zusätzlich Propagierung ähnlich dem Edge-Finding durchgeführt wird.

Abbildung 5.16 zeigt ein Beispiel, in dem Edge-Finding keine Propagierung leisten kann. Die Aufgabenmenge $S$ besteht aus den Aufgaben $A$ und $B$ mit der jeweiligen Ausführungszeit 6 und 3 und dem Ressourcenverbrauch 4 und 6. Wir nehmen in diesem Beispiel an, daß sowohl $A$ als auch $B$ bereits zum Zeitpunkt 4 plaziert sind. Die Kapazität der Ressource beträgt 10. Für Aufgabe $C$ gilt $est(C) = 1$, $lct(C) = 16$, $dur(C) = 5$ und $use(C) = 3$. Obwohl Edge-Finding hier keine Propagierung leisten kann, ist es doch offensichtlich, daß $C$ nicht vor den Aufgaben in $S$ plaziert werden kann ($start(A) < est(C) + dur(C)$). Somit kann $C$ frühestens zum Ende von $B$ beginnen, also zum Zeitpunkt 7. Solche Art von Propagierung wird von der Histogramm-Propagierung geleistet.

Wir verwenden nun für Intervalle die für Bereiche eingeführte Notation.

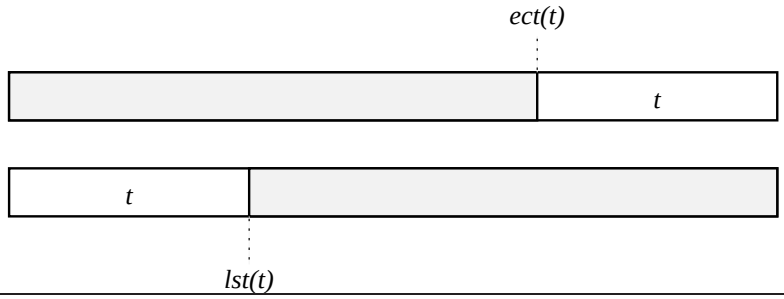
**Definition 20** *Gilt für eine Aufgabe $t$ die Ungleichung $lst(t) < ect(t)$, so wird die Ressource $res(t)$ garantiert im Intervall $lst(t)\#(ect(t)-1)$ von $t$ verbraucht (siehe Abbildung 5.17). Dieses Intervall wird* Reservierungsintervall *genannt.*

**Abbildung 5.16** Keine Propagierung durch Edge-Finding




Sind für eine Aufgabe $t$ bereits $start(t)$ und $compl(t)$ determiniert, so entspricht das Reservierungsintervall gerade dem Intervall $start(t)\#(compl(t)-1)$.

**Abbildung 5.17** Garantierter Ressourcenverbrauch einer Aufgabe $t$




Sei $S$ die Menge aller Aufgaben, die auf einer Ressource $r$ plaziert werden sollen. Sei weiterhin $O = \{t \in S | res(t) = r, lst(t) < ect(t)\}$ die Menge der Aufgaben, die ein Reservierungsintervall definieren. Dann berechnen wir die Menge $F$ aller Grenzen von Reservierungsintervallen wie folgt.

$$F = \{lst(t) | t \in O\} \cup \{ect(t) | t \in O\}$$

Nur an diesen Grenzen kann sich der garantierte Ressourcenverbrauch von Aufgaben ändern. Nun wird die Menge $P$ von Paaren $(v_1, v_2)$ berechnet, so daß $v_1$ und $v_2$ zwei direkt aufeinanderfolgende Grenzen von Reservierungsintervallen sind:

$$P = \{(v_1, v_2) | \{v_1, v_2\} \subseteq F, v_1 < v_2, \nexists v' \in F : v_1 < v' < v_2\}$$

Der Ressourcenverbrauch $use(v_1, v_2)$ zwischen $v_1$ und $v_2$, der garantiert anfällt, kann nun berechnet werden:

$$use(v_1, v_2) = \sum_{t \in O, lst(t) \leq v_1, v_2 \leq ect(t)} use(t).$$

Beachte, daß nicht $size(t)$ für die Berechnung verwendet wird, da der Ressourcenverbrauch der Reservierungsintervalle zwischen $v_1$ und $v_2$ konstant ist (aufgrund der Wahl von $v_1$ und $v_2$).

Die Zeitkomplexität der Berechnung der Werte $use(v_1, v_2)$ ist offensichtlich $O(util(r)^2)$.

Gilt $use(v_1, v_2) > cap(r)$, so kann das zugehörige Schedulingproblem keine Lösung besitzen.

Gilt für eine Aufgabe $t$, die kein Reservierungsintervall definiert, die Ungleichung $use(v_1, v_2) + use(t) > cap(r)$, so können die Werte $(v_1 - dur(t) + 1)\#(v_2 - 1)$ aus dem Bereich von $start(t)$ entfernt werden. Es ist nämlich sicherzustellen, daß $t$ überhaupt nicht zu dem Intervall $v_1\#(v_2 - 1)$ beiträgt. Entsprechende Propagierungsregeln lassen sich auch für Aufgaben formulieren, die selbst ein Reservierungsintervall definieren. Es ergibt sich Algorithmus 5.5, der für jede Aufgabe $t$ auf der Ressource $r$ prüft, ob sie in einem Intervall $v_1\#(v_2 - 1)$ für Elemente $(v_1, v_2) \in P$ plaziert werden kann. Ist dies nicht der Fall, da die Kapazität der Ressource überschritten wird, kann der Bereich von $start(t)$ reduziert werden.

**Algorithmus 5.5** Histogramm-Propagierung

```
repeat
  while P ≠ ∅ do
    waehle (v1,v2) ∈ P;
    P := P \ {(v1,v2)};
    if use(v1,v2) > cap(r) then return ⊥;
      for t ∈ S do
        if use(v1,v2) + use(t) > cap(r) and
            ( (lst(t) < ect(t) and (v1 < lst(t) or ect(t < v2)) or
              lst(t) ≥ ect(t)) then
          start(t) ∉ (v1 − dur(t) + 1)#(v2 − 1);
         end;
      end;
  end;
until kein Bereich kann mehr reduziert werden
```

Wir schreiben $start(t) \notin M$, wenn alle Werte $v \in M$ aus dem Bereich von $start(t)$ entfernt werden. Sei $k$ die maximale Kardinalität eines Bereichs von $start(t)$ für alle $t \in T$ mit $res(t) = r$.

**Satz 9** *Für die durch Algorithmus 5.5 definierte Propagierungsfunktion gelten alle Eigenschaften aus Definition 4. Der Algorithmus hat eine Laufzeitkomplexität von* $\Omega(util(r)^2)$ *bzw.* $O(k \cdot util(r)^3)$.

Beweis. Wir zeigen zuerst die Korrektheit der Propagierungsfunktion. Gilt $use(v_1, v_2) + use(t) > cap(r)$, so sind zwei Fälle zu unterscheiden. Gilt $lst(t) \geq ect(t)$, so kann $t$ nicht über ein Reservierungsintervall zu dem aktuell betrachteten Intervall $v_1\#(v_2 - 1)$ beitragen und die Bereichsreduzierung ist offensichtlich korrekt, da sich der Ressourcenverbrauch einer zu einem Reservierungsintervall beitragenden Aufgabe im Intervall selbst nicht ändert. Gilt $lst(t) < ect(t)$, so könnte $t$ zu dem aktuellen Intervall über ein Reservierungsintervall beitragen. Dies ist aber dann nicht der Fall, wenn $v_1 < lst(t)$ oder $ect(t) < v_2$ gilt, da ansonsten aufgrund der Konstruktionsvorschrift $(v_1, v_2)$ nicht in $P$ enthalten sein könnte.

Die Monotonie der Histogramm-Propagierung gilt, da die Anzahl von Reservierungsintervallen

nur größer und ein einzelnes Reservierungsintervall nur breiter werden kann. Die anderen Eigenschaften auf Definition 4 gelten offensichtlich.

Die Komplexität ergibt sich, da $|P| = O(util(r))$ und $|S| = O(util(r))$ gilt sowie der Tatsache, daß höchstens $O(k \cdot util(r))$ Elemente aus den Bereichen entfernt werden können. □

Unseres Wissens ist diese Art der Histogramm-Propagierung (also welche Intervalle für die Propagierung betrachtet werden) sonst nirgends beschrieben worden.

Für einen Projektor, der einen Kapazitätsconstraint für kumulative Schedulingprobleme realisiert, wird man sowohl den verallgemeinerten Zwei-Phasen-Algorithmus als auch Histogramm-Propagierung verwenden. Die Algorithmen können zum Beispiel so lange hintereinander ausgeführt werden, bis kein Bereich mehr reduziert werden kann.

Beachte, daß in Algorithmus 5.5 nicht nur die Werte an den Grenzen von Bereichen entfernt, sondern auch Werte im Inneren der Bereiche entfernt werden können. Dies kann für Anwendungen von Vorteil sein, die solche Informationen ausnutzen (siehe z. B. [AB93] für Plazierungsprobleme oder Kapitel 8 für eine Fallstudie). Es läßt sich jedoch auch eine Spezialisierung des Algorithmus angeben, die nur an den Grenzen von Bereichen Werte entfernt.

## 5.4 Verwandte Arbeiten

Der Begriff *Edge-Finding* wurde zwar in [AC91] geprägt, geht jedoch auf die in [CP89] eingeführten Techniken zurück. In [CP89] werden genau die Propagierungsregeln aus den Propositionen 9, 12 und 13 verwendet. Zum Edge-Finding wird nur eine Teilmenge der möglichen Aufgabenmengen herangezogen, die wie in diesem Kapitel dynamisch bestimmt wird. In der Nachfolgearbeit [CP90] wird ein Algorithmus angegeben, der die gleiche Propagierung erreicht, wie wenn alle möglichen Aufgabenmengen betrachtet würden. Die Arbeit [CP94] enthält noch einmal verbesserte Propagierungsregeln.

In [AC91] werden ähnliche Propagierungstechniken wie in [CP89] verwendet, es werden jedoch mehr Aufgabenmengen betrachtet. Zusätzlich wird in [AC91] eine andere Distribuierungsstrategie (siehe Abschnitt 6.4) verwendet als in [CP89] und es werden noch Techniken vorgestellt, wie ein guter Schedule recht schnell gefunden werden kann (siehe auch Kapitel 9).

In [CL94a] wird Edge-Finding mit Aufgabenintervallen erstmals in der Constraintprogrammierung verwendet. Der in Abschnitt 5.2.3 beschriebene Algorithmus geht auf [CL94a] zurück, berücksichtigt jedoch einige weitere Propagierungsregeln aus [CL94a] nicht, die den Algorithmus komplizierter machen, ohne viel stärkere Propagierung beizutragen (siehe auch Abschnitt 13.3). In [CL94a] wird Edge-Finding jedoch durch ein regel-basiertes System realisiert (CLAIRE [CL94b]).

Die Auswahl der Aufgabenmengen des in Abschnitt 5.2.4 geschilderten Zwei-Phasen-Algorithmus geht auf [MS96] zurück. Wir führen hier jedoch stärkere Propagierungsregeln als in [MS96] ein und verallgemeinern den Ansatz für kumulative Schedulingprobleme.

Für die Klasse der kumulativen Schedulingprobleme wird im Operations Research der Begriff *Resource-Constrained Project Scheduling* verwendet [BLK83, HDdR96, Dem92, DH92, BKST97]. In diesen Arbeiten werden nur recht schwache Propagierungstechniken verwendet,

aber es werden starke Distribuierungsstrategien entwickelt (insbesondere werden durch sogenannte *dominance rules* nur solche Zweige eines Suchbaumes betrachtet, die möglichst wahrscheinlich zu Schedules mit kleiner Länge führen).

Edge-Finding zur Lösung solcher Probleme wurde erstmals im Bereich der Constraintprogrammierung verwendet. Die Arbeit [CL96b] enthält eine Verallgemeinerung der in [CL94a] eingeführten Techniken und verwendet eine ähnliche Technik wie die in Abschnitt 5.3.2 beschriebene Histogramm-Propagierung. Auch Nuijten beschreibt in [Nui94] Algorithmen für disjunktive und kumulative Schedulingprobleme, die Edge-Finding verwenden. Die dort geschilderten Algorithmen haben das gleiche Laufzeitverhalten wie die in Abschnitt 5.2.4 und 5.3 beschriebenen, verwenden aber andere Aufgabenmengen zur Propagierung. Wir haben den Zwei-Phasen-Algorithmus zu einer konkreten Implementierung verwendet, da dieser uns früher zugänglich war als [Nui94]. Der in Abschnitt 5.3.2 beschriebene Algorithmus entstand unabhängig von [CL96b] und [Nui94]. In [Nui94] werden zur Realisierung von Edge-Finding Constraintnetze verwendet und Propagierung wird durch einen spezialisierten Algorithmus implementiert, der Kantenkonsistenz zusichert (siehe Abschnitt 2.3 oder [Mac77]). Die Techniken aus [Nui94] werden in [BPN95b] in das Constraintsystem ILOG SOLVER bzw. ILOG SCHEDULER integriert. Es wird jedoch nicht auf die konkrete Modellierung eingegangen.

Weiterführende Arbeiten im Bereich der Constraintprogrammierung sind [BLP97] und [CL97a]. Diese Arbeiten integrieren weitere Techniken aus [CL96b, Nui94, DH92, BKST97].

Weitere Arbeiten, die Constraintprogrammierung und Edge-Finding zur Realisierung von Kapazitätsconstraints verwenden sind [Col96] für disjunktive Probleme oder [Loc96] für kumulative Probleme. In [AB93] werden zwar Anwendungen von Kapazitätsconstraints im kumulativen Fall geschildert, aber keine Algorithmen vorgestellt.

# Kapitel 6

# Serialisierer

In den vorangehenden Kapiteln haben wir meistens Distribuierungsstrategien betrachtet, die eine Variable $x$ auswählen, einen Wert $l$ aus dem Bereich von $x$ bestimmen und dann z. B. mit $(\{x = l\},\ \{x \neq l\})$ distribuieren. Für disjunktive Schedulingprobleme, und nur solche werden in diesem Kapitel betrachtet, versagen diese Strategien aber oft, wenn man an guten oder optimalen Lösungen der Probleme interessiert ist (siehe auch Abschnitt 3.2.2 über das Brückenbauproblem).

Wir betrachten deshalb in diesem Kapitel Distribuierungsstrategien, die Aufgaben auf Ressourcen ordnen, indem mit geeigneten Projektoren distribuiert wird. Ein einfaches Beispiel besteht darin, zwei Aufgaben $t_1$ und $t_2$ auf einer gemeinsamen Ressource auszuwählen und mit Projektoren für die Constraints $compl(t_1) \leq start(t_2)$ und für $compl(t_2) \leq start(t_1)$ zu distribuieren. Werden auf diese Weise alle Aufgaben einer Ressource paarweise geordnet, liegt letztendlich eine Serialisierung der Aufgaben auf jeder Ressource vor (vgl. Abschnitt 5.2). Deshalb wird diese Art von Distribuierungsstrategie für disjunktive Schedulingprobleme auch *Serialisierer* genannt. Um mehrfaches Ordnen der gleichen Aufgaben effizient zu vermeiden, werden wir zustandsabhängige Distribuierungsstrategien verwenden.

Wie für Kapazitätsconstraints wollen wir auch für Serialisierer die besten verfügbaren Algorithmen verwenden. Und auch hier stammen diese Techniken aus dem Operations Research [CP89, AC91, CP90] und sind teilweise in constraintbasierte Ansätze übernommen [BPN95b, CL94a]. In diesem Kapitel werden wir diese Techniken in das Modell von Propagierung und Distribuierung einbetten. Insbesondere benötigen wir für eine effiziente Realisierung zustandsabhängige Distribuierungsstrategien. Wir werden erstmals Distribuierungsstrategien definieren, die redundante Projektoren zu Konfigurationen hinzufügen und zeigen, daß dies eine sehr erfolgreiche Technik zur Reduzierung der Suchraumgröße ist (in Abschnitt 9.4.2 erfolgt eine quantitative Analyse).

In Abschnitt 6.1 definieren wir Serialisierer. Die folgenden beiden Abschnitte befassen sich mit zwei unterschiedlichen Varianten von Serialisierern. In Abschnitt 6.2 wird eine Distribuierungsstrategie nach und nach Ressourcen serialisieren, während in Abschnitt 6.3 auch Aufgaben auf ständig wechselnden Ressourcen geordnet werden können. Das Kapitel endet mit einer kurzen Darstellung verwandter Arbeiten.

Wie für Kapazitätsconstraints findet sich eine quantitative Analyse der hier vorgestellten Techni-

ken für eine Reihe von Standardproblemen aus dem Operations Research in Abschnitt 9.4.1 und in Abschnitt 13.3.

## 6.1 Serialisierer

In diesem Kapitel betrachten wir zustandsabhängige Distribuierungsstrategien für disjunktive Schedulingprobleme. Dabei nehmen wir hier eine feste Konfiguration $(d,P,\sigma)$ an, die mit $((P_1,\sigma_1),\ldots,(P_m,\sigma_m))$ distribuiert werden soll, wobei $d$ ein Bereichsconstraint ist, $P,P_1,\ldots,P_m$ Mengen von Projektoren und $\sigma,\sigma_1,\ldots,\sigma_m$ Zustände sind (siehe Abschnitt 2.6). Sei diese Konfiguration zulässig für eine Spezifikation, die ein disjunktives Schedulingproblem beschreibt. Zur Erinnerung: Mit einer zustandsabhängigen Distribuierungsstrategie $(\Delta,\sigma_0)$ gilt

$$\{(d,P,\sigma),\ldots\} \rightarrow \{(d,P\cup P_1,\sigma_1),\ldots,(d,P\cup P_m,\sigma_m),\ldots\}$$

genau dann, wenn $\Delta(d,\sigma)=((P_1,\sigma_1),\ldots,(P_m,\sigma_m))$ gilt (siehe Definition 11). Alle anderen Reduktionsregeln wie sie in Abschnitt 2.5 für Konfigurationen beschrieben wurden, beachten den mit einer Konfiguration assoziierten Zustand nicht.

Die Distribuierungsstrategien dieses Kapitels ordnen Aufgaben auf einer Ressource. Dazu führen wir die folgenden Definitionen ein (wir übernehmen in diesem Kapitel die Notation aus Kapitel 5).

**Definition 21** *Sei eine Konfiguration $(d,P,\sigma)$ gegeben. Für zwei Aufgaben $t_1$ und $t_2$ schreiben wir $t_1 \ll t_2$ bzgl. $(d,P,\sigma)$, falls $lct(t_1)\leq est(t_2)$ gilt oder es ein $p\in P$ mit $c(p) \models\!\!\!\models compl(t_1)\leq start(t_2)$ gibt. Zwei Aufgaben $t_1$ und $t_2$ heißen* geordnet, *falls $t_1\ll t_2$ oder $t_2\ll t_1$ gilt. Wir sagen $t_1$ und $t_2$ sind* ungeordnet, *falls weder $t_1\ll t_2$ noch $t_2\ll t_1$ gilt. Wir sagen, eine Ressource $r$ ist* serialisiert, *falls alle Aufgaben auf dieser Ressource paarweise geordnet sind.*

Die hier betrachteten Distribuierungsstrategien werden mit Projektoren distribuieren, so daß nach und nach immer mehr Aufgaben geordnet sind. Wir sagen dann auch, daß die Distribuierungsstrategie *Ordnungsentscheidungen* trifft oder Aufgaben *ordnet*. Wir sagen für einen Projektor $p$ mit $c(p) \models\!\!\!\models compl(t_1)\leq start(t_2)$, daß $p$ die zwei Aufgaben $t_1$ und $t_2$ *ordnet*. Um effizient zu testen, welche Aufgaben schon geordnet sind, wird der Zustand einer Konfiguration verwendet. Wir gehen hier nicht näher auf die Struktur des Zustands ein, sondern nehmen einfach an, daß die durch eine Distribuierungsstrategie getroffenen Ordnungsentscheidungen geeignet im Zustand von Konfigurationen vermerkt werden. Dabei darf der Zustand $\sigma$ einer Konfiguration $K$ nur dann signalisieren, daß zwei Aufgaben $t_1$ und $t_2$ geordnet sind, wenn tatsächlich auch $t_1\ll t_2$ oder $t_2\ll t_1$ bzgl. $K$ gilt. Wir nehmen in diesem Kapitel an, daß der Anfangszustand $\sigma_0$ leer ist, also keine Aufgaben als geordnet ausweist. Beachte, daß, wenn aus einer Konfiguration $(d,P,\sigma)$ ein Projektor $p$ mit $c(p) \models\!\!\!\models compl(t_1)\leq start(t_2)$ durch die Subsumtionsregel in Abbildung 2.2 gelöscht wird, auf jeden Fall $lct(t_1)\leq est(t_2)$ bzgl. $d$ gilt.

**Definition 22** *Eine zustandsabhängige Distribuierungsstrategie $(\Delta,\sigma_0)$ heißt* Serialisierer, *falls für einen Distribuierungsschritt mit*

$$\Delta(d,\sigma)=((P_1,\sigma_1),\ldots,(P_m,\sigma_m))$$

*für die Konfiguration* $(d,P,\sigma)$ *folgendes gilt. Gilt*

$$\exists\{t_1,t_2\}\subseteq T, res(t_1)=res(t_2): \quad t_1 \text{ und } t_2 \text{ ungeordnet},$$

*dann gilt*

$$\forall i\in\{1,\ldots,m\}\ \exists p\in P_i: \quad c(p)\ \models\!\!\!|\ compl(t)\leq start(t'),$$

*so daß* $\{t,t'\}\subseteq T$, $res(t)=res(t')$ *und* $t,t'$ *ungeordnet bzgl.* $(d,P,\sigma)$ *sind. Ansonsten ist* $\Delta(d,\sigma)$ *beliebig.*

Gibt es also noch Aufgaben auf einer Ressource, die ungeordnet sind, wird ein Serialisierer Ordnungsentscheidungen treffen. In jedem Zweig des entstehenden Suchbaumes wird mindestens eine Ordungsentscheidung getroffen. Erst wenn alle Resourcen serialisiert sind, ist es dem Serialisierer zum Beispiel möglich, nur noch Variablen zu determinieren (z. B. durch eine first-fail Strategie). Beachte, daß in einem Distribuierungsschritt nicht alle Projektoren in einem $P_i$ Aufgaben ordnen müssen (siehe Abschnitt 6.3.1) .

Als Beispiel betrachten wir nun einen sehr einfachen Serialisierer, der in jedem Distribuierungsschritt genau zwei Aufgaben auf einer Ressource ordnet. Eine solche Distribuierungsstrategie haben wir bereits in Abschnitt 3.2 kennengelernt. Es gilt also

$$\Delta(d,\sigma)=((\{p_1\},\sigma_1),\ (\{p_2\},\sigma_2)),$$

wobei $c(p_1)$ der Constraint $compl(t_1)\leq start(t_2)$ und $c(p_2)$ der Constraint $compl(t_2)\leq start(t_1)$ ist, so daß $res(t_1)=res(t_2)$ gilt und $t_1,t_2$ ungeordnet sind. Der Zustand $\sigma_1$ wird den Projektor $p_1$ und der Zustand $\sigma_2$ den Projektor $p_2$ geeignet vermerken, um sicherzustellen, daß keine Aufgaben mehrfach geordnet werden. Für die hier betrachtete Konfiguration $K=(d,P,\sigma)$ gilt offensichtlich $d\ \wedge\ c(P)\ \models\ c(p_1)\ \vee\ c(p_2)$, da in $K$ die Kapazitätsconstraints des zugehörigen Schedulingproblems realisiert sind. Algorithmus 6.1 definiert die vorgestellte Distribuierungsstrategie $\Delta$.

**Algorithmus 6.1** Ein einfacher Serialisierer für $(d,P,\sigma)$

```
if allResourcesSerialized(d,σ) then
  return otherStrategy(d,σ);
else (t1,t2) := choosePair(d,σ);
  p1 := Projektor fuer compl(t1) ≤ start(t2);
  p2 := Projektor fuer compl(t2) ≤ start(t1);
  σ1 := addInfo(σ,p1);
  σ2 := addInfo(σ,p2);
  return ( ({p1}, σ1), ({p2}, σ2) );
end;
```

Sind alle Ressourcen serialisiert, so kann nach einer anderen Strategie weiter distribuiert werden. Ansonsten wird ein Paar von Aufgaben ausgewählt und entsprechend distribuiert. Sowohl bei der Auswahl der Aufgaben als auch bei der Ordnung des zurückgegebenen Tupels können Heuristiken eingehen, so daß vielversprechende Ordnungsentscheidungen zuerst getroffen werden.

Serialisierer können prinzipiell in zwei Klassen aufgeteilt werden. Wir werden für jede Klasse Serialisierer vorstellen, die Techniken aus dem Operations Research integrieren. *Ressourcenorientierte* Strategien wählen eine Ressource aus und serialisieren diese. Danach wird eine andere

Ressource ausgewählt und serialisiert etc. bis letztendlich alle Ressourcen serialisiert sind. *Aufgabenorientierte* Strategien müssen nicht notwendigerweise eine Ressource nach der anderen serialisieren. Oft werden diese Serialisierer in einem Distribuierungsschritt eine Ressource und zwei noch ungeordnete Aufgaben auf dieser Ressource auswählen und diese ordnen. Dies wird solange wiederholt, bis alle Ressourcen serialisiert sind.

Es werden in diesem Kapitel nur disjunktive Schedulingprobleme betrachtet, da die verwendeten Techniken nicht auf kumulative Schedulingprobleme übertragen werden können. Dort können nämlich Aufgaben gleichzeitig auf einer gemeinsamen Ressource ausgeführt werden, so daß ausschließliches Ordnen von Aufgaben in diesem Fall keine guten Ergebnisse liefert. Distribuierungsstrategien für kumulative Probleme findet man in Kapitel 8 oder in [CL96b, HDdR96, Dem92].

## 6.2 Ein ressourcenorientierter Serialisierer

In disem Abschnitt beschäftigen wir uns mit sogenannten *ressourcenorientierten Serialisierern*. Dabei werden nach und nach Ressourcen serialisiert. Im Gegensatz zum einfachen Beispiel des Algorithmus 6.1 ist es also nicht möglich, daß die Distribuierungsstrategie Aufgaben auf ständig wechselnden Ressourcen ordnet.

Der in diesem Abschnitt vorgestellte Serialisierer orientiert sich an [BPN95b] (was stark von [CP89] beeinflußt wurde). Wir zeigen wie diese Techniken in das Konzept von zustandsabhängigen Distribuierungsstrategien integriert werden können.

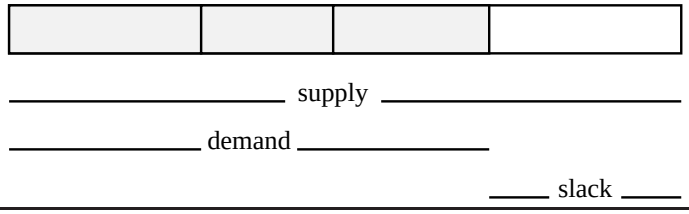
In Abschnitt 3.2 haben wir bereits eine ressourcenorientierte Strategie kennengelernt. Dort wurde die Ressource zur Serialisierung ausgewählt, für die die Summe der Ausführungszeiten ihrer Aufgaben maximal war. Dies wurde dadurch motiviert, daß diese Ressource in gewisser Hinsicht sehr *kritisch* ist, das heißt eine Art *Engpaß* (engl. *bottleneck*) des Schedulingproblems bildet. Hat das zugehörige Schedulingproblem keine Lösung, so sollte dies durch die Wahl des Engpasses möglichst schnell entdeckt werden. Hat das zugehörige Schedulingproblem jedoch Lösungen, so verbleibt auf den nicht so kritischen Ressourcen mehr 'Spielraum', die Aufgaben so zu verschieben, daß letztendlich für jede Ressource eine Serialisierung gefunden werden kann (vgl. auch die Begründung der first-fail Strategie in Abschnitt 3.1.2). Im Operations Research ist es ein übliches Vorgehen, Engpässe zuerst zu serialisieren (siehe z. B. [CP89, AC91]).

Für schwierigere Probleme als das in Abschnitt 3.2 betrachtete Brückenbauproblem ist das Kriterium, einfach die Summe der Ausführungszeiten zu betrachten, jedoch zu schwach (siehe auch Kapitel 9). Der Nachteil dieses einfachen statischen Kriteriums ist es, daß nicht dynamisch die tatsächlich verfügbare Zeit zur Plazierung der Aufgaben auf einer Ressource betrachtet wird. In der Literatur finden sich deshalb stärkere Kriterien inwieweit eine Ressource kritisch ist (siehe z. B. [CP89]). Eine Alternative ist es, den sogenannten *globalen Schlupf* (engl. *global slack*) einer Ressource zu betrachten.

**Definition 23** *Sei r eine Ressource und S die Menge aller Aufgaben $t \in T$, so daß $res(t) = r$ gilt. Dann wird $lct(S) - est(S)$ der* Vorrat *(engl.* supply*) von r gennant. Der Wert $dur(S)$ heißt die* Anforderung *(engl.* demand*) von r. Der* globale Schlupf *(engl.* global slack*) $slack_g^r$ von r ist die Differenz von Vorrat und Anforderung, also $lct(S) - est(S) - dur(S)$.*

Je kleiner der globale Schlupf, desto kritischer ist die zugehörige Ressource, da nur wenig freier Platz zum Verschieben der Aufgaben auf dieser Ressource vorhanden ist (siehe auch Abbildung 6.1). Der globale Schlupf läßt sich in $O(util(r))$ berechnen.

**Abbildung 6.1** Globaler Schlupf




Jedoch kann es Teilmengen von Aufgaben geben, für die es sehr viel weniger Verschiebungsspielraum gibt als es der globale Schlupf suggeriert. Als Beispiel betrachten wir die Aufgabenmenge $S = \{A, B, C\}$, die in der folgenden Tabelle beschrieben ist.

| Aufgabe $t$ | $start(t)$ | $dur(t)$ |
|---|---|---|
| A | 0#18 | 2 |
| B | 1#7 | 5 |
| C | 2#9 | 4 |

Der globale Schlupf von $S$ ist $lct(S) - est(S) - dur(S) = 20 - 0 - 11 = 9$. Betrachtet man jetzt jedoch das Aufgabenintervall $S' = I(B,C) = \{B,C\}$, so ergibt sich $lct(S') - est(S') - dur(S') = 13 - 1 - 9 = 3$. Das heißt, für die Aufgaben $B$ und $C$ gibt es sehr viel weniger Verschiebungsspielraum als es der globale Schlupf suggeriert. Deshalb werden wir die Bewertung einer Ressource verfeinern.

Wir definieren dazu den Begriff des *lokalen Schlupfs*. Zur Bestimmung des lokalen Schlupfs ziehen wir die Aufgabenintervalle auf einer Ressource heran. Sei dazu $I_r$ die Menge der nichtleeren Aufgabenintervalle auf $r$:

$$I_r = \{I(t_1,t_2) | \{t_1,t_2\} \subseteq T, res(t_1) = res(t_2) = r, est(t_1) \leq est(t_2), lct(t_1) \leq lct(t_2)\}$$

**Definition 24** *Sei $r$ eine Ressource. Für ein Aufgabenpaar $(t_1,t_2)$ mit $res(t_1) = res(t_2) = r$, so daß das zugehörige Aufgabenintervall $I(t_1,t_2)$ nicht leer ist, ergibt sich der* lokale Schlupf *$slack(I(t_1,t_2))$ (engl.* local slack*) des Aufgabenintervalls $I(t_1,t_2)$ wie folgt:*

$$slack(I(t_1,t_2)) = lct(t_2) - est(t_1) - dur(I(t_1,t_2)).$$

*Der* lokale Schlupf *$slack_l^r$ der Ressource $r$ ergibt sich dann zu*

$$slack_l^r = \min(\{slack(I(t_1,t_2)) | I(t_1,t_2) \in I_r\}).$$

Der lokale Schlupf kann in $O(util(r)^3)$ berechnet werden.

Es wird diejenige Ressource $r$ zuerst zur Serialisierung ausgewählt, deren lokaler Schlupf minimal ist. Ist dieser Wert bei zwei Ressourcen gleich, so wird diejenige Ressource mit dem

kleineren globalen Schlupf selektiert. Insgesamt wählt man also die Ressource, die bzgl. der lexikographischen Ordnung $(slack_l^r, slack_g^r)$ minimal ist.

Nachdem eine Ressource $r$ ausgewählt wurde, wird diese serialisiert. Jedoch müssen mit einer Strategie wie in Algorithmus 6.1, die pro Distribuierungsschritt genau zwei Aufgaben ordnet, mindestens $(util(r) \cdot (util(r) - 1))/2$ Schritte durchgeführt werden, bis $r$ serialisiert ist. Zwar kann man durch genaue Analysen einige Distribuierungsschritte in einer konkreten Implementierung vermeiden (siehe Abschnitt 6.3 und 9.4). Für große Probleme steigt allerdings auch der Rechenaufwand für eine genau Analyse stark an. Kann die Anzahl der Distribuierungsschritte nicht begrenzt werden, ist eine solche Schrittanzahl bei der Lösung von Problemen nicht realistisch (siehe auch Kapitel 10).

Wir werden hier eine Alternative betrachten, die im besten Fall sehr viel weniger Distribuierungsschritte bis zur Serialisierung benötigt. Dies wird erreicht, indem pro Distribuierungsschritt mehrere Aufgaben gleichzeitig geordnet werden. Damit steht zudem nach einem einzelnen Schritt mehr Information zur Propagierung zur Verfügung, als wenn nur ein einzelnes Aufgabenpaar geordnet würde.

Um eine Ressource zu serialisieren, kann man nach und nach jeweils eine Aufgabe auswählen und diese vor allen anderen Aufgaben auf dieser Ressource plazieren. Somit werden im günstigsten Fall nur $util(r) - 1$ Distribuierungsschritte zur Serialisierung der Ressource benötigt. Um geeignete Kandidaten auszuwählen, greifen wir auf die Ideen in Abschnitt 5.2.2 über Edge-Finding zurück.

Sei $S$ die Menge der auf $r$ auszuführenden Aufgaben. Eine Aufgabe $t \in S$ kann vor allen anderen Aufgaben in $S$ ausgeführt werden, falls

$$lct(S \backslash \{t\}) - est(t) \geq dur(S)$$

gilt.

Gibt es keine Aufgabe in $S$, für die dies gilt, so gilt $d \wedge c(P) \models\!\!\!| \perp$ für die zu distribuierende Konfiguration $K = (d, P, \sigma)$, da die realisierte Spezifikation Kapazitätsconstraints enthält, für die es somit keine Lösung geben kann. Damit kann das zugehörige Schedulingproblem keine Lösung besitzen. In diesem Fall wird die zu distribuierende Konfiguration aus der Konfigurationsmenge gelöscht; es gilt also $m = 0$ für diesen Distribuierungsschritt (siehe Abschnitt 2.6).

Sei $u(r)$ die Menge der Aufgaben auf $r$, die noch nicht mit allen anderen Aufgaben auf $r$ geordnet wurden:

$$u(r) = \{t \in T | res(t) = r, \exists t' \in T : res(t') = r, t \neq t', t' \text{ und } t \text{ ungeordnet}\}$$

Diese Menge kann aus dem Zustand $\sigma$ der zu distribuierenden Konfiguration gewonnen werden. Die Menge $F$ der Aufgaben, die vor allen anderen Aufgaben in $u(r)$ plaziert werden können, berechnet sich damit wie folgt.

$$F = \{t \in T | t \in u(r), lct(u(r) \backslash \{t\}) - est(t) \geq dur(u(r))\}$$

Es ist jetzt noch die Reihenfolge der Konfigurationen als Ergebnis eines Distribuierungsschritts zu bestimmen; also welche Projektoren in den Projektormengen $P_i$ in $((P_1, \sigma_1), \ldots, (P_m, \sigma_m))$

vorkommen. Diese Reihenfolge ist wichtig für die verwendete Suchstrategie (siehe auch Abschnitt 2.6). Da wir von Tiefensuche ausgehen, wird zuerst diejenige Konfiguration weiter reduziert, die aus $(P_1, \sigma_1)$ hervorgeht.

Wenn diejenige Aufgabe $t$ in $F$ vor allen anderen Aufgaben in $u(r)$ plaziert wird, die den kleinsten frühesten Startzeitpunkt hat (also $est(t)$), so ist der verbleibende Platz für die übrigen Aufgaben möglichst groß. Deshalb sortieren wir $F$ nach steigendem Wert von *est*. Ist dieser Wert für zwei Aufgaben gleich, sortiert man nach steigendem Wert von *lst* (also insgesamt nach der lexikographischen Ordnung bzgl. $(est(t), lst(t))$). Das Ergebnis dieses Sortierprozesses sei die Liste *Ordered*.

Die Reihenfolge der Aufgaben in *Ordered* bestimmt die Projektormengen $P_1$ bis $P_m$ mit denen distribuiert wird. Sei $o_i$ die $i$-te Aufgabe in *Ordered*, $1 \leq i \leq |F|$. Dann enthält $P_i$ diejenigen Projektoren, die die Constraints $compl(o_i) \leq start(u)$ für alle $u \in u(r) \setminus \{o_i\}$ realisieren. $P_i$ entspricht also den Constraints, daß $o_i$ vor allen anderen Aufgaben in $u(r)$ plaziert wird. Da in der zu distribuierenden Konfiguration $K = (d, P, \sigma)$ insbesondere die Kapazitätsconstraints des zugehörigen Schedulingproblems realisiert sind, gilt offensichtlich $d \wedge c(P) \models c(P_1) \vee \ldots \vee c(P_m)$ nach Konstruktion der $P_i$. Es gilt insbesondere $m = |F|$.

Algorithmus 6.2 definiert einen solchen ressourcenorientierten Serialisierer. Beachte, daß der lokale Schlupf zur Ressourcenauswahl dynamisch berücksichtigt wird und nicht eine Ressourcenreihenfolge festgelegt wird, bevor der erste Distribuierungsschritt durchgeführt wurde (wie es in dem einfachen Serialisierer aus Kapitel 3 der Fall ist). Dadurch können Engpässe besser aufgespürt werden (zu Beginn der Lösungssuche hat in der Regel erst wenig Propagierung stattgefunden, so daß Engpässe nicht so leicht zu erkennen sind; siehe auch Abschnitt 9.4.1).

Gilt $|F| = 1$, so ergibt sich nicht wirklich eine Fallunterscheidung durch Distribuierung. Stattdessen werden redundante Projektoren zur Konfiguration $K$ hinzugefügt, die distribuiert werden soll. Die Projektoren sind redundant, da sie von den Kapazitätsconstraints subsumiert werden, die durch $K$ realisiert sind. Auch wenn die Kapazitätsconstraints selbst durch keine Projektoren realisiert werden, die Edge-Finding verwenden, kommt doch durch den Serialisierer Propagierung durch Edge-Finding hinzu. Wir werden in Abschnitt 9.4 sehen, daß diese zusätzliche Propagierung durch einen Serialisierer nicht vernachlässigbar ist. In einer konkreten Implementierung bedeutet dieser Fall, daß kein Wahlpunkt angelegt werden muß. Damit kann die Suchtiefe reduziert werden, was zu weniger Speicherverbrauch führt.

**Satz 10** *Ist $k$ der maximale Wert von $util(r)$ für eine vorkommende Ressource $r$ (also $k = \max(\{util(r) | r \in R\})$), so ist die Laufzeitkomplexität mit Algorithmus 6.2 pro Distribuierungsschritt $O(|R| \cdot k^3)$ , falls eine neue Ressource zur Serialisierung ausgewählt werden muß, und $O(k)$ sonst.*

Beweis. Der lokale Schlupf für eine Ressource $r$ läßt sich in $O(util(r)^3)$ berechnen. Da bei Auswahl einer Ressource alle Ressourcen betrachtet werden müssen ergibt sich in diesem Fall eine Komplexität von $O(|R| \cdot k^3)$. Da die Menge $F$ für eine Ressource $r$ sich in $O(util(r))$ berechnen läßt, ergibt sich die zweite Behauptung. □

Analog zur Berechnung der Menge $F$ kann man auch eine Menge $L$ derjenigen Aufgaben berechnen, die nach allen anderen Aufgaben in $u(r)$ plaziert werden können. Eine Distribuierungs-

**Algorithmus 6.2** Ein ressourcenorientierter Serialisierer für $(d, P, \sigma)$

```
if Ressource r ausgewaehlt then
  if r serialisiert then
    r := selectRes(d,σ);
  end;
else
  r := selectRes(d,σ);
end;
if allResourcesSerialized(d,σ) then
  return otherStrategy(d,σ);
end;
firsts = findFirsts(d,σ);
if firsts = ∅ then return ();
end;
ordered := orderFirsts(firsts);
m := |ordered|;
P1 := Projektoren fuer {compl(ordered_1) ≤ start(u) |
                        u ∈ u(r) \ {ordered_1};
...
Pm := Projektoren fuer {compl(ordered_m) ≤ start(u) |
                        u ∈ u(r) \ {ordered_m}};
σ_1 := addInfo(σ,P1);
...
σ_m := addInfo(σ,Pm);
return ( (P1, σ_1), ..., (Pm, σ_m) );
```

strategie wird dann nur die Menge $L$ berücksichtigen (wobei dann die lexikographische Ordnung $(lct(t), ect(t))$ zur Anwendung kommt). Eine andere Distribuierungsstrategie kann sowohl $F$ als auch $L$ zur Distribuierung berücksichtigen (siehe [HMSW97]). Wir werden diese Distribuierungsstrategien in der Fallstudie in Kapitel 10 einsetzen. Wegen der relativ geringen Tiefe des entstehenden Suchbaumes (im günstigen Fall), eignen sich diese Serialisierer gut für Probleme mit vielen Aufgaben pro Ressource.

## 6.3 Ein aufgabenorientierter Serialisierer

In diesem Abschnitt wird ein aufgabenorientierter Serialisierer betrachtet. Während bei ressourcenorientierten Serialisierern nach und nach Ressourcen serialisiert werden, können bei aufgabenorientierten Serialisierern Aufgaben auf wechselnden Ressourcen geordnet werden, ohne daß eine zuvor berücksichtigte Ressource bereits serialisiert sein muß. Es wird oft mit Projektormengen $\{p_1\}$ und $\{p_2\}$ distribuiert, so daß $c(p_1)$ der Constraint $compl(t_1) \leq start(t_2)$ und $c(p_2)$ der Constraint $compl(t_2) \leq start(t_1)$ mit $res(t_1) = res(t_2)$ ist. Da in der Regel alle noch nicht geordneten Aufgabenpaare bei jedem Distribuierungsschritt betrachtet werden, besitzen diese Strategien eine größere Laufzeitkomplexität als ressourcenorientierte Strategien. Weil aber nicht immer zwei Aufgaben auf derselben Ressource betrachtet werden müssen und feiner auflösende Auswahlkriterien zum Einsatz kommen können, kann dieser Nachteil evtl. durch insgesamt weniger Distribuierungsschritte wett gemacht werden (da die getroffenen Entscheidungen zu mehr

Propagierung führen). Zudem werden wie im vorangehenden Abschnitt redundante Projektoren zu der zu distribuierenden Konfiguration hinzugefügt. Dies können jedoch sehr viel mehr redundante Projektoren sein als für ressourcenorientierte Serialisierer. In Abschnitt 9.4 wird gezeigt, daß dieser Ansatz für einige Probleme sehr erfolgreich ist, die als sehr schwierig gelten.

Die hier betrachtete Distribuierungsstrategie geht auf [CL94a] zurück. Im Gegensatz zu [CL94a] verwenden wir die Distribuierungsstrategie aber auch, um redundante Projektoren zu einer Konfiguration hinzuzufügen. Außerdem wurde das Auswahlkriterium gegenüber [CL94a] vereinfacht, ohne an Effizienz zu verlieren (siehe Abschnitt 13.3).

Ein Distribuierungsschritt dieser Strategie besteht aus zwei Teilen. Im ersten Teil wird eine Ressource selektiert, auf der zwei Aufgaben zu ordnen sind. Im zweiten Teil werden dann zwei konkrete Aufgaben ausgewählt und es wird bestimmt, mit welchen Projektoren distribuiert wird. Die Projektoren realisieren die entsprechenden Ordnungsconstraints.

### 6.3.1 Ressourcenauswahl

Zur Auswahl der Ressource $r$, auf der ein zu ordnendes Aufgabenpaar selektiert werden soll, verwenden wir eine Kombination von verschiedenen Kriterien. Dazu berechnen wir zuerst für eine Ressource $r$ den lokalen Schlupf $slack_l^r$ (siehe den vorangehenden Abschnitt).

Bei der Berechnung von $slack_l^r$ werden alle Aufgabenintervalle $I(t_1,t_2) \in I_r$ betrachtet. Gilt $slack(I(t_1,t_2)) < 0$ für zwei Aufgaben $t_1$ und $t_2$, so gilt $d \wedge c(P) \models\!\!\!| \perp$ für die zu distribuierende Konfiguration $K = (d,P,\sigma)$, da die realisierte Spezifikation Kapazitätsconstraints enthält, für die es somit keine Lösung geben kann. Damit kann das zugehörige Schedulingproblem keine Lösung besitzen. In diesem Fall wird $K$ aus der Konfigurationsmenge gelöscht; es gilt also $m = 0$ für diesen Distribuierungsschritt. Diese Tests können auch bei ressourcenorientierten Serialisierern durchgeführt werden, wo sie jedoch nur bei der Auswahl einer Ressource zum Einsatz kommen.

Gilt $slack(I(t_1,t_2)) \geq 0$ für alle Aufgaben $t_1$ und $t_2$ auf $r$, bestimmen wir nun für jedes Aufgabenintervall $I(t_1,t_2) \in I_r$ all diejenigen Aufgaben (*Kandidaten*), die bzgl. $t_1$ und $t_2$ noch nicht geordnet sind und in diesem Intervall vor ($F^r_{I(t_1,t_2)}$) bzw. nach ($L^r_{I(t_1,t_2)}$) allen anderen Aufgaben plaziert werden können.

$$F^r_{I(t_1,t_2)} = \{t \in T \mid t \in I(t_1,t_2), t \neq t_1, t \neq t_2, t \text{ und } t_1 \text{ ungeordnet}, lct(t_2) - est(t) \geq dur(I(t_1,t_2))\}$$

$$L^r_{I(t_1,t_2)} = \{t \in T \mid t \in I(t_1,t_2), t \neq t_1, t \neq t_2, t \text{ und } t_2 \text{ ungeordnet}, lct(t) - est(t_1) \geq dur(I(t_1,t_2))\}$$

Nun bestimmen wir die minimale Kardinalität beider Mengen:

$$MC(I(t_1,t_2)) = \min(\{|F^r_{I(t_1,t_2)}|, |L^r_{I(t_1,t_2)}|\})$$

Je kleiner dieser Wert, desto weniger Kandidaten gibt es, die vor (bzw. nach) allen anderen Aufgaben plaziert werden können.

Das Aufgabenintervall $I(t_1,t_2)$ auf der Ressource $r$, für das der Wert

$$slack(I(t_1,t_2)) \cdot MC(I(t_1,t_2))$$

minimal ist und in dem mindestens zwei Aufgaben noch ungeordnet sind, nennen wir $crit(r)$. Es wird jetzt diejenige noch nicht serialisierte Ressource ausgewählt, für die der Wert

$$slack(crit(r)) \cdot slack_l^r \cdot \min(\{Par, MC(crit(r))\})$$

minimal ist. Die ausgewählte Ressource bezeichnen wir als die *kritische* Ressource. Beachte, daß die Verwendung des Kriteriums $slack_l^r$ dazu führen kann, daß die Ressource mit dem kleinsten Wert von $slack_l^r$ zuerst serialisiert wird (wenn diese Ressource im Vergleich zu den übrigen einen sehr kleinen lokalen Schlupf hat). Obwohl diese Distribuierungsstrategie also aufgabenorientiert arbeitet, müssen nicht unbedingt Aufgaben auf ständig wechselnden Ressourcen geordnet werden. Der Parameter *Par* ist empirischer Natur und soll verhindern, daß Aufgabenintervalle mit sehr vielen Kandidaten in $F^r_{I(t_1,t_2)}$ bzw. $L^r_{I(t_1,t_2)}$ zuwenig Gewicht bekommen. Auch die Multiplikation der Faktoren ist empirisch begründet (in [CL94a] hat sich diese Art der Berechnung für die meisten betrachteten Probleme als der Aufsummierung überlegen gezeigt).

Bei jeder Auswahl einer Ressource werden sehr viele Aufgabenintervalle und die dazugehörigen Kandidatenmengen berechnet. Es liegt nahe, diese Information zu nutzen. Gilt zum Beispiel $F^r_{I(t_1,t_2)} = \emptyset$, so muß $t_1$ vor allen Aufgaben in $I(t_1,t_2)\backslash\{t_1\}$ plaziert werden (wegen $slack_l^r \geq 0$ gilt $lct(t_2) - est(t_1) \geq dur(I(t_1,t_2))$, so daß dies auch tatsächlich möglich ist). Wie im Abschnitt über ressourcenorientierte Strategien können in diesem Fall redundante Projektoren zu der zu distribuierenden Konfiguration $K = (d, P, \sigma)$ hinzugefügt werden. In einer konkreten Implementierung kann dies zu einer drastischen Einsparung von Wahlpunkten und damit zur Reduzierung der Suchbaumtiefe beitragen, da bei einem solchen Schritt kein Wahlpunkt in der Implementierung erforderlich ist (siehe Abschnitt 9.4.2 für eine quantitative Analyse). Im Gegensatz zum vorangehenden Abschnitt können für aufgabenorientierte Serialisierer viel mehr redundante Projektoren erzeugt werden, da viel mehr Aufgabenintervalle pro Distribuierungsschritt betrachtet werden.

Gilt also $F^r_{I(t_1,t_2)} = \emptyset$, so können Projektoren für die folgenden Constraints in einem Distribuierungsschritt zu $K$ hinzugefügt werden:

$$\forall t \in I(t_1,t_2), t \neq t_1 : compl(t_1) \leq start(t)$$

und

$$compl(t_1) \leq \min(\{lct(t_2) - (dur(I(t_1,t_2)) - dur(t_1)),\ \min(\{lst(t') | t' \in I(t_1,t_2)\backslash\{t_1\}\})\})$$

(siehe auch Abschnitt 5.2.2 über Edge-Finding).

Gilt $L^r_{I(t_1,t_2)} = \emptyset$, so können Projektoren für die folgenden Constraints zu $K$ hinzugefügt werden:

$$\forall t \in I(t_1,t_2), t \neq t_2 : compl(t) \leq start(t_2)$$

und

$$start(t_2) \geq \max(\{est(t_1) + (dur(I(t_1,t_2)) - dur(t_2)),\ \max(\{ect(t') | t' \in I(t_1,t_2)\backslash\{t_2\}\})\}).$$

Wir wollen hier annehmen, daß in einem solchen Fall $\Delta(d,\sigma) = ((P_1,\sigma_1))$ gilt, wobei $P_1$ die oben angeführten Projektoren enthält und $\sigma_1$ die entsprechenden Ordnungsbeziehungen aus $\sigma$ aktualisiert. Der Serialisierer trifft also nur Ordnungsentscheidungen, die bereits von der Konfiguration

subsumiert sind. Daß die Bedingung $d \wedge c(P) \models c(P_1)$ gilt, folgt sofort aus den Erläuterungen zu den Propagierungsregeln für Edge-Finding in Abschnitt 5.2.2, Proposition 12 und 13. Die Bevorzugung dieses Schritts gegenüber einem Ordnungsvorschlag folgt unserem Ansatz zuerst so viel Propagierung wie möglich auszunutzen, bevor der Suchraum durch Suche weiter exploriert wird. Beachte, daß Monotonie erhalten bleibt, da nur die Propositionen 12 und 13 zur Propagierung verwendet werden (siehe Satz 5).

Die Idee zu diesem Vorgehen stammt aus einem unveröffentlichten Implementierungsdetail von Yves Caseaus Schedulingansatz [CL96a]. Dort werden allerdings nur redundante Bereichsconstraints und keine Projektoren zu der zu distribuierenden Konfiguration hinzugefügt. Wir zeigen in Abschnitt 9.4.2, daß dieses Vorgehen sehr wichtig für das effiziente Lösen von Schedulingproblemen sein kann. Dies liegt daran, daß der Serialisierer starke Propagierung durch Edge-Finding beitragen kann, auch ohne daß Kapazitätsconstraints mit Aufgabenintervallen wie in Abschnitt 5.2.3 implementiert sein müssen. Caseau konnte diese Bedeutung nicht auffallen, da er zur Realisierung von Kapazitätsconstraints immer Aufgabenintervalle verwendete [CL94a, CL95].

### 6.3.2 Aufgabenauswahl

Ist keine der Kandidatenmengen leer, so wird mit einem geeigneten $((\{p_1\},\sigma_1),(\{p_2\},\sigma_2))$ distribuiert. Nachdem eine Ressource $r$ selektiert und das zugehörige Intervall $crit(r)$ bestimmt wurde, werden jetzt also zwei Aufgaben zum Ordnen ausgewählt. Dabei greifen wir auf ein häufig benutztes Kriterium zurück (siehe z. B. [BR65, ERV76, ERV80, CP89]). Wir wählen die zwei Aufgaben und diejenige Ordnung, die möglichst viel zur Reduktion von Bereichen beiträgt, das heißt, die *Entropie* (die Variabilität) des Systems möglichst stark verkleinert. Eine Approximation hierfür ist eine Abschätzung für die Bereichsreduktion der an einem Ordnungsschritt beteiligten Aufgaben. Je größer diese Reduktion im Vergleich zur Kardinalität des Bereichs ist, desto geeigneter sind die Kandidaten.

Prinzipiell sind $|crit(r)|^2$ Aufgabenpaare als zu ordnende Kandidaten zu betrachten. Um die Laufzeit der Distribuierungsstrategie nicht zu groß werden zu lassen, beschränken wir uns darauf, daß entweder $t_1$ mit einem $t \in F^r_{I(t_1,t_2)}$ oder $t_2$ mit einem $t \in L^r_{t_1,t_2}$ geordnet wird (siehe [CL94a]).

Sei $(d,P,\sigma)$ die zu distribuierende Konfiguration. Dann betrachten wir zwei zu ordnende Aufgaben $t_x$ und $t_y$ und untersuchen, welchen Einfluß die Ordnung $compl(t_x) \leq start(t_y)$ mindestens hätte. Aufgrund eines diese Ordnung realisierenden vollständigen Projektors würde die untere Grenze des Bereichs von $start(t_y)$ evtl. angehoben und die obere Grenze von $compl(t_x)$ evtl. verkleinert:

$$est(t_y) := \max(\{est(t_y), ect(t_x)\})$$
$$lct(t_x) := \min(\{lct(t_x), lst(t_y)\})$$

Die Änderung des Bereichs von $compl(t_x)$ wird mit $\delta_1(t_x,t_y)$ bezeichnet:

$$\delta_1(x,y) = lct(x) - \min(\{lct(x), lst(y)\}) = \max(\{0, lct(x) - lst(y)\})$$

Die Änderung des Bereichs von $start(t_y)$ ist $\delta_2(t_x,t_y)$ und ist analog definiert:

$$\delta_2(x,y) = \max(\{0, ect(x) - est(y)\})$$

Um die Auswirkungen der Ordnung für eine einzelne Aufgabe zu bewerten, wird die Änderung in Relation zur aktuellen Kardinalität eines Bereichs gesetzt. Hierzu definieren wir eine Funktion *evalT*. Diese Funktion wird dann z. B. wie $evalT(|dom(t_x,d)|,\delta_1(t_x,t_y))$ verwendet.

$$evalT(x,y) = \begin{cases} upper & : \text{ if } y = 0 \\ 0 & : \text{ falls } x < y \\ (x-y)^2/x & : \text{ else} \end{cases}$$

wobei *upper* der größte Wert $ect(t)$ einer Aufgabe $t \in T$ ist, also die aktuell kleinste obere Schranke für die Schedulelänge des betrachteten Schedulingproblems. Damit resultiert eine große Änderung eines Bereichs in einem kleinen Wert von *evalT*.

Um die Auswirkungen einer Ordnungsentscheidung für zwei Aufgaben (also $compl(t_1) \leq start(t_2)$ oder $compl(t_2) \leq start(t_1)$) zu bewerten, führen wir die Funktion *evalO* ein:

$$evalO(x,y) = \min(\{evalT(|dom(x,d)|,\delta_1(x,y)),\ evalT(|dom(y,d)|,\delta_2(x,y))\})$$

Das bedeutet, daß die größte Änderung eines Bereichs von $start(t)$ bzw. $compl(t)$ für eine Aufgabe $t$ (die in einem kleinen Wert von *evalT* resultiert) berücksichtigt wird.

Nehmen wir an, daß $|F^r_{crit(r)}| \leq |L^r_{crit(r)}|$ für das Aufgabenintervall $crit(r) = I(t_1,t_2)$ gilt. Der Einfluß der Ordnungsentscheidung $compl(t_1) \leq start(t)$ für ein $t \in F^r_{crit(r)}$ kann durch $evalO(t_1,t)$ ausreichend bewertet werden.

Wird aber durch $compl(t) \leq start(t_1)$ ein $t \in F^r_{crit(r)}$ vor $t_1$ plaziert, so kann sich der lokale Schlupf des Aufgabenintervalls $crit(r)$ ändern. Eine untere Schranke für diese Änderung ist

$$d(t_1,t) = \min(\{est(t) | t \in F^r_{crit(r)}\}) - est(t_1)$$

Dieser Wert sollte bei der Ordnung $compl(t) \leq start(t_1)$ auch berücksichtigt werden.

Insgesamt kann das Paar $(t_1,t)$ durch die Funktion *evalP* (z. B. mit $evalP(t_1,t)$ verwendet) wie folgt bewertet werden.

$$evalP(x,y) = \max(\{evalO(x,y),\ \min(\{evalO(y,x),\ evalT(slack(crit(r)),d(t_1,t))\})\})$$

Dasjenige Paar $(t_1,t)$ mit $t \in F^r_{I(t_1,t_2)}$ wird von der Distribuierungsstrategie zum Ordnen ausgewählt, für das der Wert von $evalP(t_1,t)$ minimal ist. Damit ist die Verringerung der Entropie maximal. Gilt $|F^r_{crit(r)}| > |L^r_{crit(r)}|$, so ist analog ein Paar $(t,t_2)$ auszuwählen (siehe Algorithmus 6.3).

Hat ein Schedulingproblem keine Lösung, so muß der gesamte Suchbaum abgesucht werden. Dies ist zum Beispiel dann der Fall, wenn man bereits die Lösung mit der optimalen Schedulelänge gefunden hat und nun beweisen möchte, daß diese Lösung tatsächlich optimal ist. Dazu schränkt man den Schedulinghorizont (siehe Seite 58) so ein, daß er um Eins kleiner als die zuvor gefundene Schedulelänge ist. Ist die zuvor gefundene Lösung optimal, hat das so definierte Schedulingproblem keine Lösung.

In diesem Fall sind also für einen Wahlpunkt alle Nachfolger im Suchbaum zu untersuchen. Deshalb ist das Aufgabenpaar zu wählen, wo beide Nachfolger einen großen Einfluß auf die Verringerung der Entropie haben. Da dies dem Fall entspricht, daß die Bewertungen für beide

Ordnungsentscheidungen möglichst klein sein sollen, wird wie oben beschrieben das Maximum beider Bewertungen für *evalP* betrachtet.

Sucht man hingegen eine (optimale) Lösung, so reicht es aus, einen Lösungsknoten zu finden, so daß im günstigsten Fall nur ein Nachfolger im Baum zu explorieren ist. Deshalb ändern wir dann die Funktion *evalP* wie folgt:

$$evalP(x,y) = \min(\{evalO(x,y),\ \max(\{evalO(y,x),\ evalT(slack(crit(r)), d(t_1,t))\})\})$$

Es wird also das Minimum der Bewertungen betrachtet.

Sei $(t_1,t_2)$ das ausgewählte Aufgabenpaar. Sei $p_1$ ein Projektor für den Constraint $compl(t_1) \leq start(t_2)$ und sei $p_2$ ein Projektor für den Constraint $compl(t_2) \leq start(t_1)$. Nun muß noch entschieden werden, ob mit dem Tupel $((\{p_1\},\sigma_1),(\{p_2\},\sigma_2))$ oder mit $((\{p_2\},\sigma_2),(\{p_1\},\sigma_1))$ distribuiert werden soll. Hat das Schedulingproblem keine Lösung, so muß der gesamte Suchbaum exploriert werden. Somit spielt die Reihenfolge im Ergebnistupel der Distribuierungsstrategie keine Rolle. Versucht man jedoch, eine Lösung zu finden, so ist es eine übliche Strategie (siehe z. B. [ERV80]), diejenige Ordnungsentscheidung zu präferieren, die für die Startzeiten der Aufgaben am meisten Verschiebungsspielraum läßt. In unserem Fall ist dies die Ordnungsentscheidung, die in einem größeren Wert von *evalO* resultiert (ähnliche Kriterien werden für Strategien verwendet, die auf Variablen- und Wertselektion beruhen; siehe z. B. [Min96]). Da wir von Tiefensuche ausgehen, ist $p_1$ derjenige Projektor, der dieser Ordnungsentscheidung entspricht.

Für den Fall $|F^r_{crit(r)}| > |L^r_{crit(r)}|$ ergeben sich ähnliche Überlegungen.

### 6.3.3 Der Gesamtalgorithmus

Insgesamt ergibt sich Algorithmus 6.3, der einen aufgabenorientierten Serialisierer definiert.

Sei $k$ der maximale Wert von $util(r)$ für eine vorkommende Ressource ($k = \max(\{util(r)|r \in R\})$).

**Satz 11** *Die Laufzeitkomplexität mit Algorithmus 6.3 pro Distribuierungsschritt ist* $O(|R| \cdot k^3)$.

Beweis. Das Berechnen aller Kandidatenmengen einer Ressource hat die Komplexität $O(k^3)$ (es gibt pro Ressource $O(k^2)$ Aufgabenintervalle mit bis zu je $k$ Elementen). Da es $|R|$ Ressourcen gibt, ergibt sich die behauptete Zeitkomplexität $O(|R| \cdot k^3)$. □

Ein Nachteil dieser Distribuierungsstrategie ist es, daß der entstehende Suchbaum im schlechtesten Fall eine Tiefe besitzt, die quadratisch in der Zahl der zu ordnenden Aufgaben wächst (nämlich $O(|R| \cdot k^2)$). Können aber viele Distribuierungsschritte durch das Hinzufügen von redundanten Projektoren ersetzt werden, so läßt sich die Tiefe des Suchbaums drastisch reduzieren. Dies ist inbesondere dann möglich, wenn die Bereiche der Variablen recht klein sind und Edge-Finding somit gute Resultate erzielen kann. Für eine quantitative Betrachtung siehe Abschnitt 9.4.

**Algorithmus 6.3** Ein aufgabenorientierter Serialisierer für $(d, P, \sigma)$

```
if allResourcesSerialized(d,σ) then
  return otherStrategy(d,σ);
else
  for r ∈ R do
    for I(t1,t2) ∈ I_r do
      F^r_I(t1,t2) := {t ∈ T | t ∈ I(t1,t2), t ≠ t1, t ≠ t2, t und t1 ungeordnet,
                       lct(t2) − est(t) ≥ dur(I(t1,t2))};
      L^r_I(t1,t2) := {t ∈ T | t ∈ I(t1,t2), t ≠ t1, t ≠ t2, t und t2 ungeordnet,
                       lct(t) − est(t1) ≥ dur(I(t1,t2))};
      if slack(I(t1,t2)) < 0 then return ()
    end;
  end;
  if ∃r ∈ R ∃{t1,t2} ⊆ T : (F^r_I(t1,t2) = ∅ ∧ ∃t' ∈ I(t1,t2)\{t1} : t1 und t' ungeordnet)
                          ∨ (L^r_I(t1,t2) = ∅ ∧ ∃t' ∈ I(t1,t2)\{t2} : t2 und t' ungeordnet)
  then P1 := finde redundante Projektoren;
    σ1 = addInfo(σ,P1);
    return ((P1,σ1))
  else
    r := findCriticalResource(d,σ);
    I(t1,t2) := crit(d,r);   // berechne crit(r)
    if |F^r_I(t1,t2)| ≤ |L^r_I(t1,t2)| then
      finde t ∈ F^r_I(t1,t2) so dass evalP(t1,t) minimal ist;
      p1 := Projektor fuer compl(t1) ≤ start(t);
      p2 := Projektor fuer compl(t) ≤ start(t1);
      σ1 := addInfo(σ,p1);
      σ2 := addInfo(σ,p2);
      if evalO(t1,t) ≥ evalO(t,t1) then
        return ( ({p1},σ1), ({p2},σ2) );
      else
        return ( ({p2},σ2), ({p1},σ1) );
      end;
    else
      finde t ∈ L^r_I(t1,t2) so dass evalP(t,t2) minimal ist;
      p1 := Projektor fuer compl(t) ≤ start(t2);
      p2 := Projektor fuer compl(t2) ≤ start(t);
      σ1 := addInfo(σ,p1);
      σ2 := addInfo(σ,p2);
      if evalO(t,t2) ≥ evalO(t2,t) then
        return ( ({p1},σ1), ({p2},σ2) );
      else
        return ( ({p2},σ2), ({p1},σ1) );
      end;
    end;
  end;
end
```

## 6.4 Verwandte Arbeiten

In [CP89] wird eine ressourcenorientierte Distribuierungsstrategie benutzt, wobei die kritischste Ressource zuerst serialisiert wird. Hierbei ist diejenige Ressource am kritischsten, deren zugehöriges Ein-Ressourcen-Problem (mit Jackson's Preemptive Schedule [Jac56] berechnet) die größte optimale Schedulelänge besitzt. Zur Serialisierung wird nach und nach die Kandidatenmenge der Aufgaben berechnet, die vor bzw. nach allen noch nicht plazierten Aufgaben plaziert werden können. Es wird die Menge mit der kleinsten Kardinalität ausgewählt. Aus dieser Menge wird dann dasjenige Aufgabenpaar selektiert, das (ähnlich wie in Abschnitt 6.3) die Entropie am meisten verringert, und dann mit entsprechenden Constraints distribuiert. In [AC91] wird auch die ressourcenorientierte Strategie aus [CP89] betrachtet.

In der Nachfolgearbeit [CP90] wird von der ressourcenorientierten Strategie aus [CP89] abgewichen. Unter Berücksichtigung des Jackson's Preemptive Schedule werden für jede noch nicht serialisierte Ressource zwei Kandidatenmengen berechnet, die Aufgaben enthalten, die zuerst bzw. zuletzt plaziert werden können. Unter diesen Mengen für alle Ressourcen wird diejenige mit der kleinsten Kardinalität ausgewählt. Ein Distribuierungsschritt besteht dann darin, ein Element aus der selektierten Menge auszuwählen und dies vor (bzw. nach) allen anderen aus dieser Menge zu plazieren.

In [BPN95b] wird ein Serialisierer verwendet, der demjenigen aus Abschnitt 6.2 entspricht und stark von [CP89] beeinflußt ist. Die Ordnungsentscheidungen der Distribuierungsstrategie aus Abschnitt 6.3 entsprechen in weiten Teilen denjenigen in [CL94a] (für Details siehe Abschnitt 6.3).

In [CS94] wird ein aufgabenorientierter Serialisierer beschrieben, der sich stark an die Arbeiten [ERV76, ERV80] anlehnt. Gleiches gilt für die Distribuierungsstrategie in [Col96]. In [Nui94] wird kein Serialisierer, sondern eine Distribuierungsstrategie beschrieben, der Variablen determiniert.

# Teil II

# Implementierung und Fallstudien

# Kapitel 7

# Propagierung und Distribuierung in Oz

In den vorangehenden Kapiteln haben wir gesehen, wie kombinatorische Probleme durch Propagierung und Distribuierung gelöst werden können. Wir haben gezeigt, wie mächtige Techniken aus dem Operations Research in Projektoren und Distribuierungsstrategien integriert werden können. Bisher haben wir aber keine konkrete Programmiersprache vorgestellt, in der dieser Rahmen implementiert ist. Genau das leistet dieses Kapitel.

Wir stellen eine Teilmenge der Sprache Oz [Smo95, HSW95, HMSW97, SSW98, Pro97] vor. Oz ist eine nebenläufige Constraintsprache, die funktionales und objektorientiertes Programmieren unterstützt und das Programmieren neuer Suchstrategien ermöglicht. Wir werden hier nur einen Teil der Sprache betrachten, um das zugrundeliegende Berechnungsmodell und die Integration von Propagierung und Distribuierung zu erklären, sowie um einige Programmbeispiele anzuführen. Wenn wir von Oz reden, meinen wir die in [Pro97, HMSW97] beschriebene Programmiersprache, die auf dem Berechnungsmodell von [Smo95] beruht (das hier vorgestellte Berechnungsmodell wird von demjenigen in [Smo95] umfaßt). Alle in dieser Arbeit aufgeführten Programme sind lauffähige Oz-Programme [Wür98].

Wir zeigen in diesem Kapitel, wie das formale Modell von Propagierung und Distribuierung in Oz implementiert werden kann. Dabei stellen wir den in Kapitel 2 eingeführten Konzepten ihre Entsprechung in Oz gegenüber. Abschnitt 7.1 führt die Teilmenge von Oz ein, die hier betrachtet wird und wir geben ein Berechnungsmodell an. Im folgenden Abschnitt gehen wir auf die Integration von Propagierung zum Lösen kombinatorischer Probleme in das Berechnungsmodell von Oz ein. In Abschnitt 7.3 werden die Designentscheidungen zur Propagierung explizit gemacht, aber auch zum Beispiel syntaktische Unterstützung oder Besonderheiten für Schedulingtechniken angeführt. Der folgende Abschnitt zeigt, wie in Oz selbst Constraints realisiert werden können. Abschnitt 7.5 enthält eine Reihe von Beispielprogrammen.

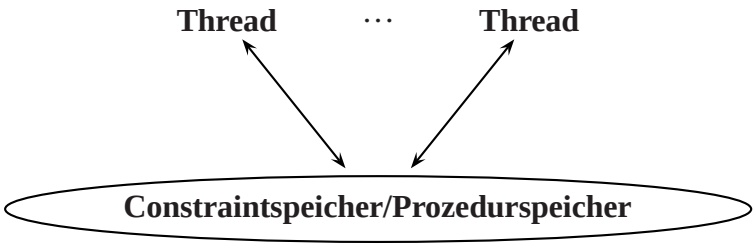
Daß wir mit der resultierenden Programmiersprache in der Lage sind, verschiedene kombinatorische Probleme effizient zu lösen, wird dann in den folgenden Kapiteln 8 bis 10 (den Fallstudien) und in Kapitel 13 (der Evaluierung) deutlich.

## 7.1 Oz

### 7.1.1 Threads

In Oz wird Berechnung durch Reduktion sogenannter *Threads* (engl. für *Kontrollfaden*) über einem gemeinsamen Speicher definiert. Der Speicher ist in einen Constraintspeicher und einen Prozedurspeicher aufgeteilt (siehe Abbildung 7.1). Threads wirken als Kontrollstrukturen für sequentielle Berechnung.

**Abbildung 7.1** Berechnung in Oz




Ein Thread enthält einen Stapel von *Anweisungen* (engl. *statements*). In einem Reduktionsschritt versucht ein Thread, die oberste Anweisung auf seinem Stapel zu reduzieren. Ein solcher Reduktionsschritt ist atomar. Anweisungen werden in *synchronisierte* und *unsynchronisierte* Anweisungen unterschieden. Eine synchronisierte Anweisung ist entweder *reduzierbar* (auch *ausführbar* genannt) oder *blockiert*. Eine blockierte Anweisung wird reduzierbar, sobald der Speicher ausreichende Information enthält, um die Anweisung zu reduzieren. Unsynchronisierte Anweisungen sind immer reduzierbar. Neue Informationen werden durch Reduktion von Threads zu dem Speicher hinzugefügt. Durch Reduktion werden auch neue Anweisungen auf den Anweisungsstapel geschrieben oder neue Threads erzeugt. Je nach dem Zustand der obersten Anweisung auf seinem Stapel ist ein Thread *reduzierbar* oder *blockiert*. Ein Thread *terminiert*, wenn sein Anweisungsstapel leer ist.

Verschiedene Threads können nebenläufig reduziert werden, das heißt, die Reduktionsstrategie muß nicht ausschließlich einen Thread bearbeiten, bis dieser terminiert oder blockiert. Die Reduktionsstrategie muß aber *Fairneß* garantieren, das heißt, jeder reduzierbare Thread wird nach endlicher Zeit reduziert. Dabei können verschiedene Threads über gemeinsame Variablen kommunizieren (Informationen austauschen) und blockierte Threads reduzierbar werden. Eine weitere Eigenschaft des Berechnungsmodells ist *Monotonie*, das heißt, ein reduzierbarer Thread bleibt auch nach der Reduktion von anderen Threads reduzierbar.

### 7.1.2 Der Constraintspeicher

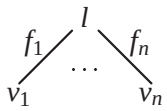
Der *Speicher* ist in den Constraintspeicher und den Prozedurspeicher aufgeteilt. Der Prozedurspeicher wird im nächsten Abschnitt erläutert. *Constraints* sind Formeln der Prädikatenlogik erster Stufe über der Signatur einer geeigneten Constraintstruktur (siehe z. B. [Smo95, SSW94, ST94]). Der *Constraintspeicher* enthält eine Konjunktion von Constraints, den sogenannten *Basisconstraints*. Basisconstraints sind hier *Gleichheitsconstraints* $x = y$ und Constraints $x = c$,

wobei $c$ ein Wert der Constraintstruktur ist (siehe unten).

Für zwei Constraints $c_1$ und $c_2$ gilt $c_1 \models c_2$, wenn die Implikation $c_1 \rightarrow c_2$ in der gewählten Constraintstruktur gültig ist. Wir sagen dann, daß $c_1$ den Constraint $c_2$ *subsumiert*.

Generell werden die Basisconstraints in der Constraintprogrammierung so gewählt, daß Erfüllbarkeit und Subsumtion effizient entschieden werden können (siehe z. B. [Smo95] oder [SSW94] für Oz oder [Wal96] für eine Übersicht). Damit sind komplexere Constraints nicht im Constraintspeicher vertreten (analog zur Aufteilung einer Spezifikationen $(d,C)$ in einen Bereichsconstraint $d$ und eine Constraintmenge $C$ in Abschnitt 2.2). Der Constraintspeicher ist immer erfüllbar.

Als Werte der Constraintstruktur nehmen wir *ganze Zahlen*, *Atome* (endliche Zeichenketten, auf denen eine totale Ordung definiert ist), *Namen* (primitive Werte ohne Struktur), *Verbunde* und die booleschen Werte **`true`** und **`false`** an. Ein *Literal* ist enweder ein Atom oder ein Name. Ein Verbund ist ein ungeordneter Baum




wobei $l$ ein Literal ist (die *Sorte*), $f_1,\ldots,f_n$ paarweise verschiedene Literale oder natürliche Zahlen sind (die *Selektoren*), und $v_1,\ldots,v_n$ Variablen sind (die *Felder*), $n > 0$.

Eine Variable $x$ heißt *determiniert*, wenn der Constraintspeicher für einen Wert $c$ den Constraint $x = c$ subsumiert. In diesem Fall sagen wir auch, daß $x$ an $c$ *gebunden* ist.

### 7.1.3 Anweisungen und Reduktionsregeln

Abbildung 7.2 enthält die Syntax von den Anweisungen der hier betrachteten Teilmenge von Oz. Bei der Reduktion des Anweisungsstapels eines Threads wird die oberste Anweisung vom Stapel genommen und gemäß den folgenden Reduktionsregeln reduziert. Dabei bedeutet die Notation $S[X/Y]$, daß in der Anweisung $S$ jedes Vorkommen von $Y$ durch $X$ ersetzt wird.

- Die sequentielle Komposition $S_1\ S_2$ schreibt zuerst $S_2$ und dann $S_1$ auf den Anweisungsstapel des Threads.
- Ein Basisconstraint als Anweisung führt dazu, daß der entsprechende Basisconstraint in den Constraintspeicher aufgenommen wird.
- Reduktion der leeren Anweisung **`skip`** entfernt diese Anweisung einfach vom Anweisungsstapel.
- Eine Deklaration **`local`** $X$ **`in`** $S$ **`end`** schreibt die Anweisung $S[Y/X]$ auf den Anweisungsstapel, wobei $Y$ eine neue Variable ist.
- Ein Konditional **`if`** $C_1$ **`then`** $S_1$ `[]` ... `[]` $C_n$ **`then`** $S_n$ **`else`** $S$ **`end`** ist eine synchronisierte Anweisung und reduziert wie folgt. $C_i$ bezeichnet den *Wächter* (engl. *guard*) des Konditionals. Subsumiert der Constraintspeicher einen Wächter $C_i$ (also die Konjunktion der

**Abbildung 7.2** Anweisungen in Oz.

| | | | |
|---|---|---|---|
| $S$ | ::= | $S_1\ S_2$ | Komposition |
| | \| | $X = K$ | Basisconstraint |
| | \| | **skip** | Leere Anweisung |
| | \| | **local** $X$ **in** $S$ **end** | Deklaration |
| | \| | **if** $C_1$ **then** $S_1$ [] ... [] $C_n$ **then** $S_n$ **else** $S$ **end** | Konditional |
| | \| | **proc** $\{X\ Y\}$ $S$ **end** | Prozedurdefinition |
| | \| | $\{X\ Y\}$ | Applikation |
| | \| | **thread** $S$ **end** | Threaderzeugung |
| $C$ | ::= | $X$= **true** \| $X$= **false** \| $C_1\ C_2$ | Klausel |
| $X, Y$ | ::= | ⟨Variablen⟩ | |
| $K$ | ::= | $X$ \| ⟨*ganzeZahl*⟩ \| ... | |

entsprechenden Basisconstraints), so wird die Anweisung $S_i$ auf den Stapel des reduzierenden Threads geschrieben. Beachte, daß dieser Schritt indeterministisch ist. Sind alle $C_i$ dissubsumiert, so wird die Anweisung $S$ auf den Stapel geschrieben. Ansonsten ist die Anweisung blockiert.

- Eine Prozedurdefinition **proc** $\{X\ Y\}$ $S$ **end** erzeugt einen neuen Namen $a$, fügt den Basisconstraint $X = a$ zum Constraintspeicher und $a : Y/S$ zum Prozedurspeicher hinzu. Der Prozedurspeicher enthält nur solche Eintragungen.
- Eine Applikation $\{X\ Y\}$ blockiert solange, bis der Constraintspeicher für einen Namen $a$ den Constraint $X = a$ subsumiert und der Prozedurspeicher die Definition $a : Z/S$ enthält. Dann reduziert die Applikation zu $S[Y/Z]$.
- Eine Threaderzeugung **thread** $S$ **end** erzeugt einen neuen Thread mit der einzigen Anweisung $S$ auf ihrem Anweisungsstapel.

Somit sind ein Konditional und eine Applikation synchronisiert. Die übrigen Anweisungen sind unsynchronisiert. Die gegebenen Definitionen für Deklaration, Prozedurdefinition und Applikation lassen sich leicht auf den Fall mehrerer deklarierter Variablen bzw. Argumente verallgemeinern.

In Abschnitt 7.1.2 wurde gesagt, daß der Constraintspeicher immer konsistent ist. Somit muß bei der Erzeugung eines Basisconstraints dafür gesorgt werden, daß vor dem Schreiben eines Basisconstraints $B$ in den Constraintspeicher $C$, die Konsistenz von $B$ mit $C$ geprüft wird. Wird dabei eine Inkonsistenz entdeckt, so wird eine Ausnahmebehandlung aufgerufen. Entsprechendes gilt auch, wenn zum Beispiel bei einer Applikation {X Y} die Variable X an ein Atom gebunden ist. Mehr Information hierüber (im Kontext mit Bereichsconstraints) findet sich in Abschnitt 7.2.4.

Addition und Subtraktion stehen über entsprechende Infixnotation zur Verfügung. Eine Prozedur für die Addition `Z = X+Y` wird so lange blockieren, bis X und Y an ganze Zahlen gebunden

sind und wird dann den Basisconstraint `Z` $= c$ in den Constraintspeicher schreiben, wobei $c$ die Summe der Werte von `X` und `Y` ist. Wie Gleichungen und Ungleichungen als Constraints realisiert werden können, wird in Abschnitt 7.3.2 beschrieben.

Eine besondere Art von Verbund ist das *Tupel*, bei dem die Selektoren nur aus ganzen Zahlen zwischen 1 und $n$ bestehen, z. B. `f(1:X 2:Y)`. Solche Tupel können zu `f(X Y)` abgekürzt werden. Eine *Liste* ist entweder das Literal `nil` oder ein zweistelliges Tupel mit ´|´ als Sorte, dessen Argumente der Listenkopf und der Listenrest sind. Listen können auch in Infixnotation geschrieben werden. Zum Beispiel steht `X|Y|nil` für das Tupel `´|´(X ´|´(Y nil))`. Listen, die mit `nil` abgeschlossen sind, können auch in der Form `[...]` geschrieben werden. Auf die Werte eines Verbundes kann mit der *Selektionsanweisung* zugegriffen werden. Zum Beispiel bindet `X = f(a:1 b:2).a` die Variable `X` an `1`.

Als weitere syntaktische Variation soll die *funktionale Schachtelung* betrachtet werden. In Oz kann eine Prozedur, deren letzter formaler Parameter immer funktional benutzt wird, als Funktion geschrieben werden (für Details siehe [Hen97b]). Zum Beispiel kann die Quadrierfunktion

```
proc {Square X Y} Y=X*X end
```

auch als Funktion

```
fun {Square Y} X*X end
```

geschrieben werden. Applikationen von Funktionen (bzw. funktionalen Prozeduren) können geschachtelt benutzt werden. Zum Beispiel ist

```
Z = {Square X} + {Square Y}
```

eine Abkürzung für

```
local XX YY in
   {Square X XX}  {Square Y YY}
   Z = XX + YY
end
```

Eine alternative Schreibweise hierfür ist

```
local
   XX={Square X}  YY={Square Y}
in
   Z = XX + YY
end
```

### 7.1.4 Berechnungsräume und Distribuierung

Berechnung in Oz findet in sogenannten *Berechnungsräumen* statt. Ein Berechnungsraum enthält den Prozedur- und Constraintspeicher, eine Menge von Threads und evtl. wieder weitere Berechnungsräume. Es ergibt sich eine Hierarchie von Berechnungsräumen, woraus sich einige sehr interessante Programmiertechniken ergeben (siehe zum Beispiel [SSW94, Sch98b]). So ist es zum Beispiel möglich, beliebige Prozeduraufrufe in den Wächtern von Konditionalen zu verwenden

(anstatt nur die Basisconstraints wie hier vorgestellt), indem die Wächter des Konditionals durch untergeordnete Berechnungsräume realisiert werden. Durch die dadurch erreichte Kompositionalität ist es zum Beispiel möglich, Inferenzmaschinen in Inferenzmaschinen ablaufen zu lassen [Sch98b].

Wegen der fairen Behandlung von Threads werden auch Anweisungen in Berechnungsräumen reduziert, die in Threads von untergeordneten Berechnungsräumen enthalten sind.

Im obersten Berechnungsraum der Hierarchie (auch *Toplevel* genannt) läuft die übliche funktionale bzw. prozedurale Berechnung ab. Untergeordnete Berechnungsräume erben den Constraintspeicher des direkt übergeordneten Berechnungsraumes.

Der Aufruf einer Ausnahmebehandlung in einem Berechnungsraum führt dazu, daß dieser Berechnungsraum *fehlschlägt*. Ein Berechnungsraum heißt *gelöst*, falls alle enthaltenen Threads terminiert und alle enthaltenen Berechnungsräume gelöst sind. Ein Berechnungsraum heißt *stabil*, wenn kein Thread mehr reduzierbar und der Berechnungsraum weder fehlgeschlagen noch gelöst ist und dies auch für alle enthaltenen Berechnungsräume gilt.

Eine weitere Anweisung in Oz ist die (unäre) *Wahlanweisung*

**`choice`** $S$ **`end`**

beziehungsweise die binäre Wahlanweisung

**`choice`** $S_1$ `[]` $S_2$ **`end`**

Diese Anweisungen sind synchronisiert. Unäre Wahlanweisungen blockieren den sie enthaltenden Thread. Wird ein Berechnungsraum stabil und enthält einen Thread, der als oberste Anweisung auf seinem Stapel die Anweisung **`choice`** $S$ **`end`** enthält, so wird diese Anweisung durch die Anweisung $S$ ersetzt. Damit wird durch die unäre Wahlanweisung auf Stabilität synchronisiert.

Enthalte ein vom Toplevel verschiedener Berechnungsraum $Sp$ einen Thread mit **`choice`** $S_1$ `[]` $S_2$ **`end`** als oberster Anweisung auf seinem Anweisungsstapel. Ist $Sp$ stabil geworden, wird der Berechnungsraum $Sp$ dupliziert. Sei $Sp'$ das Duplikat. Nun hängt es von der gewählten Suchstrategie ab, welcher Berechnungsraum zuerst weiter betrachtet wird. Für (linksorientierte) Tiefensuche, die wir in dieser Arbeit annehmen, ist dies $Sp$. In $Sp$ wird dann die Wahlanweisung durch $S_1$ ersetzt, so daß der enthaltende Thread wieder reduzierbar wird. Schlägt $Sp$ später fehl, so wird in $Sp'$ die Wahlanweisung durch $S_2$ ersetzt und der enthaltende Thread wird reduzierbar.

In Oz können neue Suchstrategien über entsprechende Abstraktionen vom Benutzer programmiert werden [Sch97b]. Eine Reihe von Suchstrategien stellt Oz bereits über vordefinierte Prozeduren zur Verfügung [HMSW97]. Dies sind Strategien zum Finden einer ersten Lösung, aller Lösungen oder auch Branch-and-Bound. Alle diese Suchstrategien erhalten als Eingabe eine einstellige Prozedur, die das zu lösende Problem beschreibt; das sogenannte *Skript*. Das Argument dieses Skripts wird als *Lösungsvariable* bezeichnet. Die Suchprozeduren geben die Bindung der Lösungsvariablen in einer Lösung zurück. Branch-and-Bound erhält als weitere Eingabe eine Prozedur, die Lösungen bewertet und miteinander vergleicht. Wird eine Lösung gefunden, so wird in alle Berechnungsräume, die weder gelöst, noch fehlgeschlagen sind, ein Thread injiziert. Dieser Thread enthält die Applikation der oben erwähnten Bewertungsprozedur. In der Regel

wird diese Prozedur dafür sorgen (durch Propagierer; siehe Abschnitt 7.2.3), daß die Bewertung der noch zu findenden Lösungen besser sein muß als die Bewertung der zuvor gefundenen Lösung (siehe auch Abschnitt 3.2). Mehr Information zum Thema Suche steht in [HMSW97] und in [Sch97b, SSW94, SS94, Sch98b].

## 7.2 Integration von Constraints zur Lösung kombinatorischer Probleme

In diesem Abschnitt beschreiben wir die prinzipielle Integration von Constraints zum Lösen kombinatorischer Probleme in Oz. Dazu werden wir das formale Modell aus Kapitel 2 geeignet in Oz einbetten. Im folgenden Abschnitt werden wir dann auf mehr konkrete Designentscheidungen eingehen. Die prinzipielle Integration von Constraints ist in [SSW94] zuerst beschrieben worden.

### 7.2.1 Der Constraintspeicher

Zur Integration des Modells aus Kapitel 2 werden wir zuerst Bereichsconstraints in den Constraintspeicher aufnehmen.

Der Constraintspeicher enthält nun auch Basisconstraints $x \in \delta$, wobei $\delta \subseteq \mathbb{Z}$, endlich und nichtleer ist (siehe Abschnitt 2.2). In Oz wollen wir auch einen einzelnen Constraint $x \in \delta$ *Bereichsconstraint* nennen.

Wir können die Definitionen und Eigenschaften aus Kapitel 2 unter Berücksichtigung von Gleichheitsconstraints geeignet übernehmen. So ist in einem Constraintspeicher $c$ der Bereich einer Variablen $x$ die kleinste Menge $\delta$, so daß $c \models x \in \delta$ gilt. Beachte, daß $\models$ nun logische Subsumtion bezeichnet; für zwei Constraints $c_1$ und $c_2$ gilt $c_1 \models c_2$, wenn die Implikation $c_1 \rightarrow c_2$ in der gewählten Constraintstruktur gültig ist. Damit können wir für Berechnung in Oz nur von Konfluenz modulo logischer Äquivalenz sprechen.

Eine Variable $x$ heißt eine *FD-Variable* (FD steht für finite domain), falls der Constraintspeicher einen Bereichsconstraint $x \in \delta$ enthält.

### 7.2.2 Anweisungen für Bereichsconstraints

Für unsere Integration betrachten wir hier nur, wie Bereichsconstraints über die Reduktion von Anweisungen in den Constraintspeicher aufgenommen werden. Propagierer werden im nächsten Abschnitt behandelt.

Bereichsconstraints $x \in \delta$ stehen über die Infixanweisung `X :: D` zur Verfügung, wobei $D$ wie in Abbildung 7.3 spezifiziert ist. Die Syntax von Bereichsconstraints entspricht also derjenigen aus Kapitel 2. Auch diese Infixanweisung wird auf entsprechende vordefinierte Prozeduren zurückgeführt. In Oz können FD-Variablen nicht beliebige ganze Zahlen als Werte annehmen, sondern nur Werte zwischen Null und einer implementierungsabhängigen Konstante *sup*. Für DFKI Oz, der konkreten Implementierung von Oz am DFKI in Saarbrücken (siehe Abschnitt 7.6), ist der

Wert von *sup* 134 217 726 (siehe auch Abschnitt 11.2). Mit `compl` kann man das Komplement eines Bereichs spezifizieren (z. B. bedeutet `compl(3#8)` die Menge $[0\#2\ \ 9\#sup]$).

**Abbildung 7.3** Bereichsconstraints `X :: D`

$$D ::= S \mid \texttt{compl}(S)$$

$$S ::= i \mid [I^+]$$

$$I ::= i \mid i\#i$$

$$i ::= \langle \text{ganze Zahl } \in \{0,\ldots,sup\}\rangle$$

### 7.2.3 Propagierer

Ein *Propagierer* ist eine konkrete Implementierung eines Projektors $p$ aus Kapitel 2, der einen Constraint $c$ realisiert. Der Propagierer implementiert also sowohl die Propagierungsfunktion $N_p$ als auch die Subsumtionsfunktion $E_p$. Als Erweiterung von Projektoren berücksichtigen Propagierer auch Gleichheitsconstraints, die in Oz Basisconstraints sind. Wir sagen auch von einem Propagierer, daß er *c realisiert*. Die *Argumente* eines Propagierers sind diejenigen FD-Variablen, die die Argumente des Constraints $c$ modellieren.

Propagierer sind in Oz als nebenläufige Agenten realisiert und werden durch entsprechende vordefinierte Prozeduren erzeugt, die teilweise durch Infixanweisungen verfügbar sind (siehe Abschnitt 7.3.2). Die vordefinierten Prozeduren werden in einem eigenen Thread reduziert, der nach Erzeugung des Propagierers terminiert. Wir nehmen an, daß ein Propagierer nach seiner Erzeugung immer dann ausgeführt werden soll, wenn für eines seiner Argumente ein Basisconstraint zum Constraintspeicher hinzugefügt wurde (also ein Gleichheitsconstraint oder ein Bereichsconstraint; siehe auch Abschnitt 2.5 und Kapitel 11).

Um zu propagieren, fügt ein Propagierer Basisconstraints zum Constraintspeicher hinzu. Dies entspricht der Propagierungsregel aus Abschnitt 2.5. Erkennt ein Propagierer, daß dies zu einer Inkonsistenz des Speichers führen würde (weil $c$ vom Constraintspeicher dissubsumiert ist), *schlägt* der Propagierer *fehl* und hört auf zu existieren; dies entspricht der ersten Fehlerregel für Konfigurationen (siehe auch Abschnitt 7.2.4). Auch wenn ein Propagierer erkennt, daß $c$ subsumiert ist, hört er auf zu existieren. In diesem Fall sagen wir, daß der Propagierer *subsumiert* ist (dies entspricht der Subsumtionsregel für Konfigurationen in Abschnitt 2.5).

Propagierer haben eine große Ähnlichkeit mit Threads. Im Gegensatz zu Threads haben Propagierer aber keine weitere innere Struktur und können nicht wie Threads in ihrer Ausführung unterbrochen werden; ihre Ausführung ist also *atomar*. Wie ein Thread wird ein Propagierer nebenläufig ausgeführt.

Wir sagen ein Constraintspeicher ist in *Normalform* bzgl. der vorhandenen Propagierer, wenn keine Propagiererausführung mehr zu einem stärkeren Constraintspeicher führen kann (siehe auch Abschnitt 2.5). Auch die anderen Begriffe und Eigenschaften für Konfigurationen aus Abschnitt 2.5 können auf Constraintspeicher und Propagierer übertragen werden.

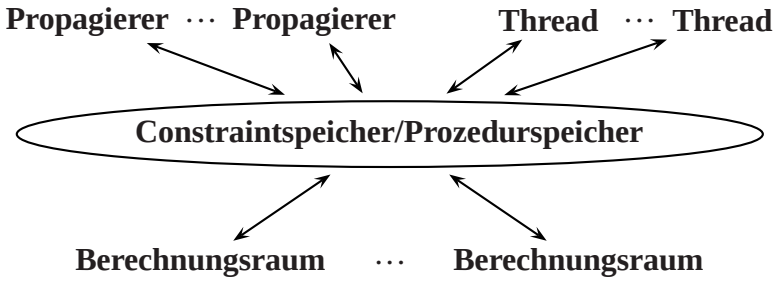
Die Fairneßeigenschaft aus Abschnitt 7.1.1 muß auch für Propagierer gelten. Da die Ausführung jedes Propagierers endlich ist, kann die Ausführung von Propagierern mit einer fairen Threadreduktion verbunden werden, so daß jeder Propagierer, der ausgeführt werden soll, nach endlicher Zeit auch ausgeführt wird. Damit sorgt die faire Reduktionsstrategie von Oz dafür, daß die Normalform des Constraintspeichers bzgl. der vorkommenden Propagierer in endlicher Zeit berechnet wird.

Durch die Definition geeigneter Prozeduren hat ein Programmierer die Möglichkeit, Threads zu erzeugen, die sich wie Propagierer verhalten. Solche Prozeduren werden als *benutzerdefinierte Propagierer* oder *benutzerdefinierte Constraints* bezeichnet. Oz stellt geeignete Operationen zur Verfügung, um benutzerdefinierte Propagierer zu ermöglichen (siehe auch [SSW94]). Ein Beispiel ist in Abschnitt 7.5.4 zu finden; siehe auch die Diskussion in Abschnitt 7.4.

### 7.2.4 Berechnungsräume und Distribuierung

Für unsere Integration ergänzen wir die Definition von Berechnungsräumen, indem diese jetzt auch noch Propagierer enthalten können. Damit wird ein Berechnungsraum zu einer Erweiterung des Konzepts einer Konfiguration aus Abschnitt 2.5. Der sich ergebende Aufbau eines Berechnungsraumes ist in Abbildung 7.4 gezeigt.

**Abbildung 7.4** Aufbau eines Berechnungsraumes




Wegen der nun enthaltenen Propagierer müssen wir auch einige der Definitionen für Berechnungsräume ergänzen. Der Aufruf einer Ausnahmebehandlung in einem Berechnungsraum und das Fehlschlagen eines enthaltenen Propagierers führt dazu, daß dieser Berechnungsraum *fehlschlägt*; dies entspricht der zweiten Fehlerregel aus Abschnitt 2.5. Ein Berechnungsraum *Sp* heißt *blockiert*, wenn der Berechnungsraum nicht fehlgeschlagen ist, der Constraintspeicher in Normalform bzgl. der im Berechnungsraum enthaltenen Propagierer ist, kein Thread mehr reduzierbar und dies auch für alle enthaltenen Berechnungsräume gilt. Ein Berechnungsraum *Sp* heißt *stabil*, falls für jede Verstärkung eines Constraintspeichers in einem *Sp* übergeordneten Berechnungsraum der Raum *Sp* blockiert bleibt und der Constraintspeicher von *Sp* nicht inkonsistent würde. Ein Berechnungsraum heißt *gelöst*, falls der Berechnungsraum stabil ist sowie alle Propagierer subsumiert und alle enthaltenen Threads terminiert sind und dies auch für die untergeordneten Berechnungsräume gilt.

Eine Distribuierungsstrategie aus Abschnitt 2.6 kann in Oz durch eine Prozedur implementiert

werden, die Wahlanweisungen verwendet. Solche Prozeduren heißen *Distribuierer*. Will man einen Distribuierungsschritt mit $(P_1, P_2)$ realisieren, wobei $P_1$ und $P_2$ Projektormengen sind, so muß die Anweisung $S_1$ bzw. $S_2$ Aufrufe von entsprechenden Propagierern für $P_1$ bzw. $P_2$ enthalten. Die Reduktion einer Wahlanweisung implementiert damit die Distribuierungsregel für Konfigurationen aus Abbildung 2.2. Auf diese Weise kann ein Suchbaum erzeugt werden, wie er in Abschnitt 2.6 beschrieben wurde. Ein Wahlpunkt wird nur für binäre Wahlanweisungen angelegt (nur hier muß ein Berechnungsraum kopiert werden).

Beachte, daß in Oz auch mehrere Distribuierer hintereinander aufgerufen werden können, die insgesamt eine einzige Distribuierungsstrategie implementieren.

## 7.3 Design des FD-Systems von Oz

In den vorangehenden Abschnitten wurden die Grundkonzpte von Oz vorgestellt und es wurde gezeigt, wie Bereichsconstraints, Propagierer und Distribuierer in das Berechnungsmodell von Oz integriert sind. In diesem Abschnitt wird diese Integration konkretisiert. Wir betrachten grundlegende Designziele und -entscheidungen zur Entwicklung der Funktionalität zum constraintbasierten Lösen von kombinatorischen Problemen in Oz. Betrachtet werden Benutzbarkeit, Kompatibilität mit den übrigen Sprachkonzepten von Oz und das Synchronisationsverhalten von Propagierern. Außerdem stellen wir die verfügbare Syntax für Propagierer vor, gehen auf Funktionalität zum Schreiben von benutzerdefinierten Propagierern und Distribuierern ein und zeigen die Punkte auf, die bei der Integration der Schedulingtechniken aus Kapitel 5 und 6 besonders zu berücksichtigen waren.

Im folgenden wird von dem *FD-System* von Oz gesprochen. Dies umfaßt die Basisconstraints im Constraintspeicher, die vordefinierten Propagierer und die vordefinierten Prozeduren zum Schreiben von benutzerdefinierten Propagierern und Distribuierern. Weiterhin ist damit auch die Implementierung durch eine abstrakte Maschine einbegriffen (siehe hierzu Kapitel 11).

Die Funktionalität des FD-Systems ist über den Verbund `FD` verfügbar, der eine Bibliothek von vordefinierten Propagierern und Distribuierern bereitstellt. Die einzelnen Prozeduren und Propagierer werden in entsprechenden Feldern zur Verfügung gestellt. Teilweise steht diese Funktionalität auch über Infixnotation zur Verfügung.

### 7.3.1 Grundlegendes Design

Die wichtigsten Designziele bei dem Entwurf des FD-Systems von Oz waren:

- *Expressivität*, um möglichst viele Constraints und Anwendungsprobleme zu realisieren,
- *Effizienz*, um große und schwierige Anwendungen aus der Praxis zu lösen,
- *Kompatibilität* mit den übrigen Sprachkonzepten von Oz.

Die Punkte *Expressivität* und *Effizienz* wurden bereits ausführlich in Kapitel 4 bis 6 diskutiert. In Oz werden die meisten der dort vorgestellten Projektoren als Propagierer zur Verfügung gestellt

(für eine genaue Beschreibung des FD-Systems siehe [HMSW97, SSW98]). Dies gilt insbesondere für die in Kapitel 5 vorgestellten Projektoren für Kapazitätsconstraints. Auch sämtliche Distribuierungsstrategien aus Kapitel 2 und die Serialisierer aus Kapitel 6 sind verfügbar.

Das Ziel der *Kompatibilität* mit Oz und einer guten Benutzbarkeit führte zu folgenden Designentscheidungen.

- Das Verhalten von Propagierern muß die Basisconstraints von Oz berücksichtigen; das bedeutet insbesondere die Berücksichtigung von Gleichheit zwischen Variablen. So sollte ein Propagierer für den Constraint, daß $n$ Variablen paarweise verschiedene Werte annehmen müssen, fehlschlagen, wenn zwei seiner Argumente zur Laufzeit gleichgesetzt werden. Ein Propagierer für eine Gleichung wie $3 \cdot X + 3 \cdot Y = Z$ sollte das Propagierungsverhalten von einem Propagierer für $6 \cdot Y = Z$ zeigen, wenn zur Laufzeit der Constraint $X = Y$ in den Constraintspeicher aufgenommen wird.

- Es muß einer Hierarchie von Berechnungsräumen Rechnung getragen werden (siehe Abschnitt 7.1.4). Dies hat Konsequenzen für die Implementierung (siehe Kapitel 11).

- In Oz wird Distribuierung durch das explizite Kopieren von Berechnungsräumen realisiert (siehe [Sch98b]). Deshalb ist es wichtig, daß Probleme sich durch möglichst wenige Propagierer repräsentieren lassen, um den Speicherverbrauch gering zu halten. Somit werden in Oz viele symbolische oder globale Constraints unterstützt (siehe Abschnitt 4.2), die durch einzelne Propagierer realisiert sind. Insbesondere wird ein Constraint für eine Gleichung oder Ungleichung durch jeweils einen einzelnen Propagierer realisiert. Anstatt, daß die Implementierung eine sehr große Zahl von Propagierern mit jeweils unterschiedlicher Stelligkeit zur Verfügung stellt, gibt es Propagierer, die mit einer beliebigen Zahl von Argumenten (z. B. in Listen enthalten) ausgeführt werden können und jeweils den entsprechenden Projektor implementieren (siehe Abschnitt 11.4).

- Die vordefinierten Prozeduren, die einen Propagierer erzeugen, laufen in einem eigenen Thread ab. Dieser Thread blockiert in der Regel bis für alle Argumente des Propagierers, die FD-Variablen sein müssen, auch tatsächlich Bereichsconstraints im Speicher vorhanden sind (und die Argumente, die Atome oder ganze Zahlen sein müssen, determiniert sind). Im allgemeinen ist nämlich das Fehlen von Bereichsconstraints ein Programmierfehler. Die Ausnahme von dieser Regel sind die funktional schachtelbaren Propagierer (siehe unten).

- Da Oz funktionale Notation unterstützt, können einige Propagierer (bzw. ihre erzeugenden Prozeduren) funktional geschachtelt verwendet werden (wie z. B. `{FD.plus X Y Z}` für den Constraint $X + Y = Z$). Dabei wird die Variable, die das Funktionsergebnis bestimmt, automatisch als FD-Variable mit dem Bereich 0#*sup* deklariert. Dies gilt auch für schachtelbare Infixnotation für reifizierte Constraints (siehe Abschnitt 7.5.1).

- Da alle in Oz implementierten Distribuierungsstrategien Wahlanweisungen verwenden, wird erst dann distribuiert, wenn ein Berechnungsraum stabil geworden ist (wie es in Abschnitt 2.5 und 2.6 gefordert wird). Damit wird sichergestellt, daß für den Distribuierungsschritt soviel Information wie möglich ausgenutzt wird.

### 7.3.2 Syntaktische Unterstützung

Infixanweisungen ermöglichen das bequeme Schreiben von Constraints und reifizierten Constraints. Ein Beispiel für Infixanweisungen für Bereichsconstraints wurde bereits in Abschnitt 7.2.2 vorgestellt. Für Variablen in einer Liste, können Bereichsconstraints mit Hilfe des Infixoperators `:::` zum Constraintspeicher hinzugefügt werden.

Weitere Infixanweisungen stehen für Gleichungen und Ungleichungen zur Verfügung. Es werden dann die entsprechenden vordefinierten Prozeduren aufgerufen. Daß es sich um durch Propagierer realisierte Constraints handelt, wird durch das Anhängen eines Doppelpunktes an die üblichen Relationssymbole verdeutlicht. Die Propagierer implementieren die Projektoren, wie sie in Abschnitt 4.1 beschrieben wurden. Die möglichen Relationsymbole $\sim_R$ und das jeweils entsprechende Relationssymbol `R` in Oz sind in der folgenden Tabelle abgebildet.

| `R` | `=:` | `>:` | `>=:` | `<:` | `=<:` | `\=:` |
|---|---|---|---|---|---|---|
| $\sim_R$ | $=$ | $>$ | $\geq$ | $<$ | $\leq$ | $\neq$ |

Die Ausdrücke in den Gleichungen bzw. Ungleichungen können die Operatoren `+`, `-`, `*`, `~`, `(` und `)` enthalten. Insbesondere sind auch nicht-lineare Ausdrücke erlaubt. Im Gegensatz zu Systemen wie ECL$^i$PS$^e$ [ECR96], in denen die realisierenden Propagierer blockieren bis der Ausdruck linear wird, wird in Oz sofort propagiert.

Geschachtelte Infixanweisungen für Constraints führen zur Erzeugung eines reifizierten Constraints (siehe Abschnitt 4.4). Zum Beispiel kann die Reifikation von $X < Y$ in die 0/1-Variable $Z$ (also $(X < Y) \leftrightarrow (Z = 1)$) durch

```
Z = X<:Y
```

ausgedrückt werden. Beachte, daß der Propagierer für den reifizierten Constraint automatisch den Constraint `Z` $\in$ `0#1` zum Constraintspeicher hinzufügt. Beispiele für reifizierte Constraints sind in Abschnitt 7.5 zu finden.

### 7.3.3 Reflektion

Sollen benutzerdefinierte Propagierer oder neue Distribuierer programmiert werden, muß der Programmierer Bereiche von Variablen reflektieren und Veränderungen des Constraintspeichers beobachten können. Dazu kann mit den Prozeduren in den Verbunden `FD.reflect` und `FD.watch` Information über den aktuellen Bereich einer Variablen reflektiert werden.

Zum Beispiel wird durch `{FD.reflect.min X I}` der kleinste Wert des Bereichs von `X` an `I` gebunden. Beachte, daß diese Operation nicht monoton ist. So könnte für einen stärkeren Constraintspeicher die Variable `I` eventuell an einen anderen Wert gebunden werden als für einen schwächeren Constraintspeicher (deshalb sollten solche Operationen erst bei Stabilität eines Berechnungsraumes ausgeführt werden). Entsprechend kann man auch den größten Wert eines Bereichs oder seine Kardinalität reflektieren.

Einige weitere Prozeduren erlauben es, den Bereich einer FD-Variablen zu beobachten und bei einem bestimmten Ereignis, eine Variable an einen booleschen Wert zu binden. Zum Beispiel

blockiert `{FD.watch.min X 10 B}` bis der Constraintspeicher `X` $\in$ `11#FD.sup` subsumiert und bindet dann `B` an **`true`** (`FD.sup` ist der Wert *sup* aus Abbildung 7.3). Subsumiert der Constraintspeicher `X` $\in$ `0#10`, so wird `B` an **`false`** gebunden. Ähnliche Prozeduren gibt es, um das Maximum und die Kardinalität eines Bereichs zu beobachten. Diese Prozeduren können bei der Implementierung von benutzerdefinierten Propagierern (Abschnitt 7.4) Verwendung finden.

### 7.3.4 Scheduling in Oz

Kapazitätsconstraints aus Kapitel 5 und Serialisierer aus Kapitel 6 sind in Oz als Propagierer bzw. Distribuierer vorhanden. In Oz bezeichnen wir die Distribuierer, die Serialisierer implementieren, auch einfach als Serialisierer. Wir zeigen hier die Punkte auf, die bei der Integration der Schedulingtechniken besonders berücksichtigt werden mußten oder von den zuvor geschilderten Verfahren abweichen. Beachte, daß die in Oz realisierten Propagierer für Kapazitätsconstraints noch leicht stärkere Propagierung anbieten als sie in Kapitel 5 geschildert wurde (es wird zusätzlich die Propagierung durchgeführt, die durch die Realisierung eines Kapazitätsconstraints durch reifizierte Constraints erhalten wird; siehe Abschnitt 5.2.1). Um die Anzahl der verwendeten FD-Variablen gering zu halten, wird in Oz nur die Startzeit $start(t)$ einer Aufgabe $t$ modelliert und als Argument von Propagierern für Kapazitätsconstraints und von Serialisierern verwendet. Die Fertigstellungszeit $compl(t)$ wird durch den Ausdruck $start(t) + dur(t)$ modelliert. Aufgrund der in Kapitel 5 gemachten Annahmen geht dadurch jedoch keine Propagierung verloren.

**Kapazitätsconstraints** Wir haben in Kapitel 5 gesehen, daß einige der Propagierungsfunktionen für Kapazitätsconstraints nicht monoton sind. Damit kann Konfluenz in dem hier vorgestellten Berechnungsmodell nicht mehr sichergestellt werden. Um die zugrundeliegenden Algorithmen trotzdem für Propagierer verwenden zu können, wird in der Implementierung von Oz die Ausführung von Propagierern für Kapazitätsconstraints so lange verzögert, bis alle übrigen Propagierer des gleichen Berechnungsraumes ausgeführt sind. Erst dann werden die Propagierer für Kapazitätsconstraints ausgeführt. Deren Ausführung geschieht dann immer in der gleichen Reihenfolge. Insgesamt kann somit letztendlich, durch Vorkehrungen in der Implementierung, Konfluenz (modulo logischer Äquivalenz) doch sichergestellt werden (siehe Abschnitt 11.7.3).

In [CL94a] werden die Datenstrukturen, die die Aufgabenintervalle enthalten, inkrementell aktualisiert, indem einer Bereichsänderung immer eine Änderung der entsprechenden Aufgabenintervalle vorangeht. Dies ist in einem Spezialsystem wie CLAIRE [CL94b] gut machbar. In Oz könnte man Inkrementalität erreichen, indem von allen verwendeten Propagierern (also auch für Gleichungen etc.) Spezialversionen implementiert werden, die vor jeder Bereichsänderung die Aufgabenintervalle aktualisieren. Damit geht aber Modularität verloren, da jeder zusätzliche Constraint wieder erst in der Spezialversion vorliegen muß. Als Alternative könnte der Kapazitätspropagierer zu Beginn nur diejenigen Aufgabenintervalle aktualisieren, deren Elemente sich geändert haben (siehe auch Abschnitt 11.5.1). Aus Einfachheitsgründen wurde darauf jedoch verzichtet und die Aufgabenintervalle werden bei jedem Aufruf des Propagierers neu berechnet. Der Zwei-Phasen-Algorithmus berechnet zwar auch die betrachteten Aufgabenintervalle immer neu, jedoch ist dies wegen der insgesamt geringeren Laufzeitkomplexität nicht so laufzeitkritisch (siehe insbesondere auch die Evaluierung in Abschnitt 13.3).

**Serialisierer** Da in CLAIRE [CL94a] auch die Aufgabenintervalle für Serialisierer inkrementell aktualisiert werden, müssen bei der Distribuierung keine Aufgabenintervalle mehr neu berechnet werden. In Oz wird die Neuberechnung hingegen bei jedem Aufruf der Distribuierung durchgeführt (aus dem gleichen Grund wie für Kapazitätsconstraints). Im Gegensatz zu [CL94a] werden in Oz nicht nur redundante Basisconstraints sondern redundante Propagierer zum Berechnungsraum hinzugefügt (Abschnitt 9.4 enthält eine genauere Analyse hierzu).

In Kapitel 6 wurden Serialisierer beschrieben, die redundante Projektoren zur Konfiguration hinzufügen, die distribuiert werden soll. Die in Oz verfügbaren Serialisierer fügen statt der Projektoren nun Propagierer zum Berechnungsraum hinzu. Im Unterschied zu Abschnitt 6.3 fügen die in Oz implementierten aufgabenorientierten Serialisierer aber in einem einzigen Schritt evtl. anfallende redundante Propagierer zum Berechnungsraum hinzu *und* distribuieren mit Propagierern für eine Ordnungsentscheidung (beachte daß dieses Vorgehen immer noch korrekt ist, da die redundanten Propagierer vom aktuellen Constrainspeicher und den restlichen Propagierern im Berechnungsraum subsumiert werden). Insbesondere wird die mögliche Propagierung durch die redundanten Propagierer nicht bei der Auswahl der Ordnungsentscheidung berücksichtigt. Diese Designentscheidung wurde aus Effizienzgründen getroffen. Pro Distribuierungsschritt bei aufgabenorientierten Serialisierern müssen nämlich alle Aufgabenintervalle neu berechnet werden. Zudem resultiert das in Abschnitt 6.3 geschilderte Vorgehen nicht notwendigerweise in einer kleineren Zahl von notwendigen Ordnungsentscheidungen (für eine genaue Analyse siehe Abschnitt 9.4.2).

In Kapitel 6 wurden ressourcenorientierte Serialisierer vorgestellt, die evtl. mit $m > 2$ Projektormengen distribuieren. Offensichtlich kann ein solcher Distribuierungsschritt auch mit binären Wahlanweisungen realisiert werden. In der linken Anweisung der Wahlanweisung erfolgt dann das Plazieren einer Aufgabe vor allen übrigen Aufgaben auf der Ressource (siehe Abschnitt 6.2). In der rechten Anweisung wird der Distribuierer rekursiv für die übrigen Kandidaten aufgerufen. Sind keine keine Kandidaten mehr übrig, wird der Berechnungsraum fehlschlagen, da keine Lösung des zugehörigen Schedulingproblems existieren kann.

Da Serialisierer zustandsabhängige Distribuierungsstrategien sind, werden die getroffenen Ordnungsentscheidungen geeignet in dem Oz-Programm vermerkt, das den Serialisierer implementiert. Wie ein solcher Serialisierer effizient implementiert werden kann, wird in Abschnitt 11.7.5 beschrieben.

## 7.4 Benutzerdefinierte Propagierer

Prinzipiell können alle in dieser Arbeit vorgestellten Propagierer in Oz selbst programmiert werden. Dazu ausreichend sind die Prozeduren zur Reflektion und Beobachtung von Bereichen (in den Verbunden `FD.reflect` und `FD.watch`) in Verbindung mit Threads und Konditionalen. Es muß jedoch darauf geachtet werden, daß die Information über Bereiche durch Propagierung nicht invalidiert wird (für Propagierer, die Kapazitätsconstraints realisieren, könnte dies zum Beispiel falsche Schlüsse durch Edge-Finding nach sich ziehen). Im Gegensatz zu den atomar ausgeführten vordefinierten Propagierern, kann ein in Oz implementierter und in einem eigenen Thread ausgeführter benutzerdefinierter Propagierer nämlich unterbrochen werden, so daß an anderer Stelle mit der Berechnung fortgefahren wird. Unter Umständen kann dies dann die Bereiche von

Argumenten des benutzerdefinierten Propagierers ändern. Dieses Problem kann umgangen werden, indem der benutzerdefinierte Propagierer zu Beginn der Berechnung einen Schnapschuß von den aktuellen Bereichen der Variablen macht (durch Reflektion in geeignete Datenstrukturen). Die Propagierungsalgorithmen rechnen dann auf diesem Schnapschuß.

Jedoch ist die Effizienz von benutzerdefinierten Propagierern oft nicht ausreichend und der Speicherverbrauch oft zu hoch. Aus diesem Grund sind die vordefinierten Propagierer auch nicht in Oz programmiert, sondern stehen in der abstrakten Maschine von Oz als C++-Programme zur Verfügung (siehe Kapitel 11).

Ein Beispiel soll diesen Effizienzunterschied verdeutlichen. Dazu betrachten wir den inkonsistenten Constraint $x < y \wedge y < x$ mit dem Bereich 1#100000 für $x$ und $y$. Auf einer SPARC20 mit 70 MHz und DFKI Oz, Version 2.0.4, werden etwa 0.5 Sekunden und 10 KB Speicher benötigt, um die Inkonsistenz durch Propagierung zu entdecken. Verwendet man hingegegen eine Prozedur ähnlich wie in Programm 7.3, so erhöht sich die Laufzeit auf 11.5 Sekunden und der Speicherverbrauch auf der Halde auf 23 MB.

## 7.5 Beispiele

Ein typisches Skript (siehe Abschnitt 7.1.4) hat folgende Struktur (in Oz wird das Prozentzeichen als Kommentarzeichen verwendet).

```
proc {TypicalScript Solution}
   local
      % Deklariere Variablen
   in
      % Bereichsconstraints und Propagierer
      ...
      % Distribuierer
   end
end
```

Zuerst werden Variablen deklariert und über die Lösungsvariable `Solution` zugänglich gemacht. Anschließend werden Bereichsconstraints und Propagierer erzeugt. Das Programm endet meist mit dem Aufruf eines geeigneten Distribuierers.

Da Skripte first-class sind, können sie bei der Programmierung von parametrisierbaren Problemlösern Verwendung finden. Ein Beispiel finden wir in Kapitel 9, in dem ein sogenannter Scheduling-Compiler Skripte erzeugt, die über verschiedene Propagierer und Distribuierer parametrisiert sind.

### 7.5.1 Reifizierte Constraints

Mit reifizierten Constraints können Constraints über den Wahrheitswerten von Constraints realisiert werden (siehe Abschnitt 4.4). Zum Beispiel kann man den Constraint, daß es höchstens zwei Variablen in der Liste `Xs` geben darf, deren Bereich eine Teilmenge von 5#9 ist, wie folgt ausdrücken.

```
fun{InD X} X::5#9 end
{FD.sum {Map Xs InD} ´=<:´ 2}
```

Dabei realisiert ein Propagierer `{FD.sum Ys ´=<:´ 2}` den Constraint $\texttt{Ys}_1 + \ldots + \texttt{Ys}_n \leq 2$. Die Funktion `Map` wendet die Funktion `InD` auf jedes Element von `Xs` an. Haben zum Beispiel zwei Listenelemente den Bereich 6#8, so werden die Bereiche der übrigen Variablen zu `compl(5#9)` reduziert.

Eine weitere Anwendung ist die Modellierung logischer Verknüpfungen zwischen Constraints. Will man zum Beispiel ausdrücken, daß Aufgabe *X* vor Aufgabe *Y* beendet sein muß oder umgekehrt, so kann dies mit reifizierten Constraints wie folgt modelliert werden (wir nehmen an, daß die Ausführung jeder Aufgabe 10 Zeiteinheiten benötigt).

```
(X + 10 =<: Y) + (Y + 10 =<: X) =: 1
```

Reifizierte Constraints lassen sich auch benutzen, um Probleme zu bearbeiten, bei denen nicht alle Constraints gleichzeitig erfüllbar sind (siehe Kapitel 8 zu sogenannten weichen Constraints).

### 7.5.2 Ein Zahlenpuzzle

Programm 7.1 enthält ein Oz-Programm für das Zahlenpuzzle aus Abschnitt 3.1.1 (gesucht sind verschiedene Ziffern für die Buchstaben *S*, *E*, *N*, *D*, *M*, *O*, *R*, *Y*, so daß die Gleichung $SEND + MORE = MONEY$ gilt). Wir übernehmen das Modell aus Abschnitt 3.1.1 und verwenden den Projektoren entsprechende Propagierer. Nachdem Bereichsconstraints für die vorkom-

**Programm 7.1** Ein Zahlenpuzzle

```
proc {Money Sol}
   local
      S E N D M O R Y
   in
      Sol = [S E N D M O R Y]
      Sol ::: 0#9
      {FD.distinct Sol}
      S \=: 0
      M \=: 0
      1000*S + 100*E + 10*N + D
      + 1000*M + 100*O + 10*R + E
      =: 10000*M + 1000*O + 100*N + 10*E + Y
      {FD.distribute naive Sol}
   end
end
```

menden Variablen erzeugt wurden, wird der Propagierer `FD.distinct` erzeugt, der besagt, daß die Ziffern paarweise verschieden sein müssen. Die Ziffern `S` und `M` müssen von Null verschieden sein. Anschließend wird die Gleichung mit Hilfe eines einzelnen Propagierers realisiert. Die mit dem Parameter `naive` selektierte Distribuierungsstrategie, wählt die am weitesten links vorkommende nicht-determinierte Variable in `Sol` aus und selektiert den kleinsten Wert in deren Bereich.

Die erste Lösung `[9 5 6 7 1 0 8 2]` des Problems kann mit

```
S = {SearchOne Money}
```

gefunden werden. Der entstehende Suchbaum ist in Abbildung 3.1 gezeigt.

### 7.5.3 Das *n*-Damen-Problem

Programm 7.2 zur Lösung des *n*-Damen-Problems lehnt sich an die Modellierung in Abschnitt 3.1.2 an. Hierbei wird `L1N` an die Liste `[1 2 ... N]` und `LM1N` an die Liste `[~1 ~2 ... ~N]` gebunden. Die Variable `Board` ist eine Liste der Länge `N` von FD-Variablen mit dem Bereich 1#*N*. Anschließend werden die drei Propagierer erzeugt, die die Constraints realisieren. Dabei realisiert

**Programm 7.2** Das *n*-Damen-Problem

```
proc {Queens N Board}
   local
      L1N  = {List.number 1 N 1}
      LM1N = {List.number ~1 ~N ~1}
   in
      Board = {FD.list N 1#N}
      {FD.distinct Board}
      {FD.distinctOffset Board LM1N}
      {FD.distinctOffset Board L1N}
      {FD.distribute ff Board}
   end
end
```

`{FD.distinctOffset [`$X_1$ $X_2$ `...` $X_n$`] [`$C_1$ $C_2$ `...` $C_n$`]}`

den Constraint, daß $X_1 + C_1, \ldots, X_n + C_n$ paarweise verschieden sein müssen. Am Ende des Programms wird die first-fail Distribuierungsstrategie mit der Liste der FD-Variablen aufgerufen. Diese Strategie unterscheidet sich von der naiven, indem die in `Board` am weitesten links stehende nicht-determinierte Variable mit dem kleinsten Bereich zuerst selektiert wird.

Die folgende Funktion berechnet ein Skript zur Lösung des Problems für `N` Damen.

```
fun {QueensSolve N}
   local Q in
      proc {Q Board}
          ...
      end
      Q
   end
end
```

Dabei ist die Prozedur `Q` wie in Programm 7.2 definiert, wobei jedoch `N` nun ein Parameter der Funktion `QueensSolve` ist, die die Prozedur `Q` als Ergebnis hat. Alle Lösungen für das 5-Damen-Problem kann man mit

```
S = {SearchAll {QueensSolve 5}}
```

erhalten (S ist dann an `[[1 3 5 2 4] [1 4 2 5 3] ...]` gebunden).

### 7.5.4 Ein benutzerdefinierter Propagierer

In diesem Abschnitt soll exemplarisch gezeigt werden, wie ein benutzerdefinierter Propagierer programmiert werden kann. Als Beispiel dient der Constraint $X \leq Y$, der durch die Prozedur `Leq` realisiert wird (siehe Programm 7.3).

**Programm 7.3** Ein Propagierer für $X \leq Y$

```
proc{Leq X Y XLow YHigh}
   local
      BX = {FD.watch.min X XLow}
      BY = {FD.watch.max Y YHigh}
   in
      if BX = true then
         local New = {FD.reflect.min X}
         in
            Y :: New#FD.sup
            {Leq X Y New YHigh}
         end
      [] BY = true then
         local New = {FD.reflect.max Y}
         in
            X :: 0#New
            {Leq X Y XLow New}
         end
      else skip
      end
   end
end
```

Die Prozedur `Leq` ist über die zwei Variablen `X` und `Y`, sowie über die aktuelle untere Bereichsgrenze `XLow` von `X` und die obere Grenze `YHigh` von `Y` parametrisiert. Zuerst werden die Variablen `BX` und `BY` eingeführt, die an **`true`** gebunden werden, sobald `X` größer als `XLow` bzw., sobald `Y` kleiner als `YHigh` wird. In diesem Fall kann das Konditional reduzieren, schränkt den Bereich der jeweils anderen Variable entsprechend ein und die Prozedur wird rekursiv mit der neuen Bereichsgrenze aufgerufen. Sind beide Variablen determiniert, so wird der else-Zweig des Konditionals mit der leeren Anweisung ausgeführt. Der Propagierer ist dann subsumiert.

Beachte, daß dieser benutzerdefinierte Propagierer in einem eigenen Thread appliziert werden muß, damit das Konditional nicht die übrige Berechnung blockiert. Weiterhin wird nicht auf Gleichheit zwischen `X` und `Y` geprüft und Subsumtion könnte früher entdeckt werden (nämlich wenn die obere Grenze des Bereichs von `X` kleiner oder gleich der unteren Grenze des Bereichs von `Y` ist).

### 7.5.5 Ein einfacher Distribuierer

Programm 7.4 enthält ein Beispiel für einen Distribuierer, der der Strategie `{FD.distribute naive Xs}` entspricht. Die Applikation `{`==` X Y Z}` blockiert bis der Constraintspeicher genügend Information enthält, um zu entscheiden ob `X` und `Y` gleich sind. Je nach Ergebnis wird `Z` an **`true`** oder **`false`** gebunden. Entsprechend für die Applikation `{`>` X Y Z}`.

**Programm 7.4** Ein einfacher Distribuierer

```
fun {Select Xs}
   local BEqual = {`==` Xs nil} in
      if BEqual = true then nil
      else
         local
            XSize    = {FD.reflect.size Xs.1}
            BGreater = {`>` XSize 1} in
            if BGreater = true then Xs.1
            else {Select Xs.2}
            end
         end
      end
   end
end
proc {Naive Xs}
   choice
      local   X = {Select Xs}
         BEqual = {`==` X nil} in
         if BEqual = true then skip
         else
            choice X = {FD.reflect.min X}
               {Naive Xs}
            [] X \=: {FD.reflect.min X}
               {Naive Xs}
            end
         end
      end
   end
end
```

Die Funktion `Select` selektiert die am weitesten links stehende Variable in der Liste `Xs`, die noch nicht determiniert ist. Sind alle Variablen determiniert, ist `nil` das Funktionsergebnis.

Die Prozedur `Naive` wartet auf Stabilität des Berechnungsraumes (mit der unären Wahlanweisung). Dann wird mit Hilfe von `Select` eine Variable selektiert. Sind alle Variablen determiniert, wird die leere Anweisung ausgeführt. Ansonsten wird ein binärer Wahlpunkt erzeugt, in dem die selektierte Variable `X` entweder an die aktuelle untere Grenze $n$ des Bereichs von `X` gebunden wird oder aber dieser Wert aus dem Bereich entfernt wird. Das Problem wird also entweder mit $\{\mathtt{X} = n\}$ oder mit $\{\mathtt{X} \neq n\}$ distribuiert. Die Distribuierungsprozedur wird dann wieder rekursiv aufgerufen.

## 7.6 DFKI Oz

Das Programmiersystem DFKI Oz [Pro97], das am DFKI in Saarbrücken entwickelt wurde, ist eine konkrete Implementierung des Berechnungsmodells von Oz [Smo95]. DFKI Oz ist weit umfangreicher als der Ausschnitt, der in diesem Kapitel gezeigt wurde. Insbesondere unterstützt DFKI Oz auch objektorientiertes Programmieren und das Erstellen von graphischen Benutzerschnittstellen. Beides wird in den Kapiteln 8–10 bei der Vorstellung der Fallstudien eine Rolle spielen.

# Kapitel 8

# Erstellen von Stundenplänen

In diesem Kapitel wird ein Projekt zur Erstellung eines wöchentlichen Stundenplans für eine Fachhochschule vorgestellt [HW96b, HW96a]. Im Rahmen dieses Projekts wurde in Oz das erste größere Problem durch Constraintprogrammierung gelöst.

Dieses Problem stellt bereits hohe Anforderungen an die Expressivität eines FD-Systems. Unerläßlich zur Modellierung sind reifizierte Constraints, um zum Beispiel Überschneidungen von Kursen auszuschließen oder, um die Zahl von Kursen zu bestimmten Zeitpunkten quantifizieren zu können. Die Einführung sogenannter weicher Constraints oder eine auf das Problem zugeschnittene Distribuierungsstrategie ermöglichen es, schnell eine relativ gute Lösung zu finden (gemessen an einem bestimmten Gütekriterium). Die Zahl der verwendeten Propagierer und Variablen kann durch die Verwendung von globalen Constraints drastisch reduziert werden. In Kapitel 12 werden wir sehen, wie das Problem durch die Verwendung von sogenannter konstruktiver Disjunktion noch besser gelöst werden kann.

## 8.1 Problemstellung

Im Frühjahr 1995 erhielten wir durch den Prorektor der Katholischen Hochschule für Soziale Arbeit in Saarbrücken die Möglichkeit, ein Constraintprogramm zu erstellen, das für alle im Sommer vertretenen Semester einen festen Wochenstundenplan berechnet. Dabei war die Zuordnung von Vorlesungen zu Lehrern bereits im Vorstadium entschieden.

Dieses Problem ist eigentlich ein gemischtes Scheduling- und Zuteilungsproblem (siehe Abschnitt 5.1). Für die einzelnen Vorlesungen müssen geeignete Räume und ein dazu passender Zeitplan gefunden werden. Da es jedoch für die Räume keine weiteren Constraints gibt, kann das Zuteilungsproblem erst einmal vernachlässigt werden. Solange zugesichert wird, daß zu jedem Zeitpunkt nicht mehr Räume benutzt werden als vorhanden sind, kann nach Berechnung eines Zeitplans die Raumverteilung trivial vorgenommen werden. Somit bilden Räume gleicher Größe eine Ressource mit einer Kapazität, die der Anzahl dieser Räume entspricht. Damit reduziert das Problem auf ein reines Schedulingproblem.

Die Hochschule bietet ein Vierjahresprogramm für einen Abschluß an. Manche Vorlesungen werden nur für ein bestimmtes Semester angeboten, andere können von Studenten aller Semester be-

sucht werden. Es gibt Pflichtveranstaltungen und solche, die freiwillig sind. Da Vorlesungen in verschiedene Gruppen aufgeteilt werden können, sprechen wir auch von Kursen. Insgesamt gibt es 91 Kurse, 34 Lehrer und 7 Räume unterschiedlicher Größe. Jeder Kurs muß in einem Raum geeigneter Größe (oder einem größeren) stattfinden, wobei zwei große, drei mittelgroße und zwei kleine Räume vorhanden sind. Es wird an 5 Schultagen unterrichtet und jeder Kurs muß zwischen $8.^{15}$ Uhr und $17.^{00}$ Uhr stattfinden. Die Kurse können jeweils zu einer vollen Viertelstunde beginnen. Es gibt Kurse von kurzer Dauer (45 Minuten) und Kurse von langer Dauer (90 Minuten). Nach einem kurzen Kurs muß es mindestens eine Pause von 15 Minuten und nach einem langen Kurs eine Pause von mindestens 30 Minuten geben.[1] Es gibt im Sommer nur Studierende des zweiten, vierten, sechsten und achten Semesters.

Zusätzlich muß ein Stundenplan die folgenden Bedingungen erfüllen.

C1) Manche Kurse können nur zu bestimmten Zeiten stattfinden.
C2) Manche Lehrer können nicht zu jeder Zeit unterrichten.
C3) Es gibt für die verschiedenen Semester unterschiedliche Zeiten für die Mittagspause.
C4) Die Kurse des 6. Semesters müssen mittwochs stattfinden.
C5) Manche Kurse müssen zur gleichen Zeit stattfinden.
C6) Manche Kurse müssen nacheinander stattfinden.
C7) Manche Kurse müssen am gleichen Tag stattfinden.
C8) Manche Kurse müsen an verschiedenen Tagen stattfinden.
C9) Manche Lehrer können nicht zur gleichen Zeit unterrichten.
C10) Zu jedem Kurs muß ein Raum geeigneter Größe vorhanden sein.
C11) Ein Lehrer darf nur einen Kurs gleichzeitig unterrichten.
C12) Die Vorlesungen eines Semesters dürfen sich im allgemeinen nicht überschneiden.
C13) Zwei Vorlesungen eines Semesters dürfen sich jedoch überschneiden, falls sie in Gruppen aufgeteilt sind (und weitere Ausnahmen).
C14) Manche freiwillige Kurse dürfen sich nicht mit Kursen des zweiten und vierten Semesters überschneiden; manche dürfen sich mit gar keinen anderen Kursen überschneiden.
C15) Ein Lehrer darf nicht mehr als 6 Stunden pro Tag unterrichten.
C16) Ein Lehrer darf nicht mehr als drei Tage in der Woche unterrichten.

Der Stundenplan sollte die folgenden Kriterien so gut wie möglich erfüllen. Die Vorlesungen des zweiten und vierten Semesters sollten möglichst montags, dienstags oder mittwochs stattfinden. Ist dies nicht möglich sollten sie donnerstags oder freitags vormittags stattfinden. Das achte Semester sollte seine Vorlesungen am Dienstag oder Donnerstag besuchen können, das sechste Semester die seinen am Mittwoch. Die Zahl der Kurse nach der Mittagspause und freitags sollte möglichst gering sein.

[1] Zusätzlich gibt es noch Block- und Wochenendkurse. Diese werden aber getrennt vom Hauptstundenplan geplant.

## 8.2 Modell

In diesem Abschnitt wird ein Modell für das Problem eingeführt.

Die kleinste unterschiedene Zeiteinheit ist eine Viertelstunde. Da die Kurse zwischen $8.^{15}$ Uhr und $17.^{00}$ Uhr stattfinden, gibt es pro Tag 36 theoretische Anfangszeiten. Bei fünf Schultagen macht dies insgesamt $5 \cdot 36 = 180$ Anfangszeiten. Damit kann die Startzeit $start(C)$ eines Kurses $C$ durch eine Variable repräsentiert werden, für die der Bereichsconstraint $start(C) \in 1\#180$ gilt. Für die Dauer eines Kurses $C$ schreiben wir $dur(C)$. Für das Ende eines Kurses verwenden wir den Ausdruck $start(C) + dur(C)$ in Constraints (in der konkreten Implementierung spart diese Variante Speicherplatz, beeinflußt aber nicht die Propagierung). Da die Kurse bis spätestens $17.^{00}$ Uhr beeendet sein müssen, können zu Beginn einige Werte aus den Bereichen entfernt werden. Zum Beispiel kann für einen Kurs $C$ mit einer Dauer von $D$ Viertelstunden der Constraint

$$start(C) \notin [36 - D\#36 \quad 72 - D\#72 \quad 108 - D\#108 \quad 144 - D\#144 \quad 180 - D\#180]$$

verwendet werden.

Kann ein Kurs $C$ zum Beispiel nur montagnachmittags stattfinden, so kann dies durch den Constraint $start(C) \in 19\#36$ modelliert werden. Kann ein Lehrer nicht montag- und dienstagvormittags unterrichten, läßt sich dies durch $start(C) \notin [1\#18\ 37\#54]$ modellieren. Entsprechend können die Constraints C1 bis C4 durch Constraints umgesetzt werden, die durch vollständige Projektoren realisiert sind. Constraint C5 läßt sich durch einfache Gleichheit zwischen den Startzeiten modellieren.

Die schwierigeren Constraints C6 bis C16 werden durch Projektoren realisiert, deren Propagierungsstärke den in Kapitel 4 angeführten Projektoren entspricht. Muß ein Kurs $C_1$ zum Beispiel vor einem Kurs $C_2$ stattfinden (Constraint C6), so kann dies durch einen Projektor für

$$start(C_1) + dur(C_1) \leq start(C_2)$$

realisiert werden. Zur Realisierung der weiteren Constraints wird zumindest Reifikation benötigt.

Wir betrachten jetzt Constraint C7, wobei der betreffende Tag nicht im voraus bekannt ist. Die Tatsache, daß ein Kurs $C$ an einem bestimmten Tag (durch den Bereich *Day* repräsentiert) stattfindet, kann mit Hilfe eines reifizierten Constraints festgestellt werden. Ist *Courses* die Menge der Kurse, die am gleichen Tag stattfinden sollen, so kann diese Menge durch

$$S = \{B \mid (start(C) \in Day) \leftrightarrow B = 1,\ C \in Courses\}$$

auf eine Menge $S$ aus 0/1-wertigen Variablen abgebildet werden. Beachte, daß es wichtig ist, den Constraint $start(C) \in Day$ zu verwenden und nicht den Constraint $start(C) \geq D_1 \wedge start(C) \leq D_2$, der durch die Projektoren für

$$\begin{gathered}(start(C) \geq D_1) \leftrightarrow B_1 = 1 \\ (start(C) \leq D_2) \leftrightarrow B_2 = 1 \\ B_1 + B_2 = 2\end{gathered}$$

realisiert werden kann, wobei $D_1$ und $D_2$ die untere bzw. obere Grenze von *Day* sind. Die Projektoren für die letzteren Constraints berücksichtigen bei ihrer Propagierung nämlich keine Lücken in Bereichen.

Eine weitere 0/1-wertige Variable *BDay* reflektiert nun den Constraint, daß mindestens ein Kurs am Tag *Day* stattfindet:

$$(\sum_{s \in S} s \ \geq 1) \ \leftrightarrow \ BDay = 1$$

Auf diese Weise verfährt man für jeden Wochentag und erhält so 0/1-wertige Variablen *BMo* bis *BFr*. Constraint C7 kann dann durch einen Projektor für

$$BMo + BTu + BWe + BTh + BFr = 1$$

realisiert werden. Constraint C8 kann durch einen Projektor für

$$BMo + BTu + BWe + BTh + BFr = |Courses|$$

realisiert werden, wenn *Courses* die entsprechenden Kurse enthält. Offensichtlich kann Constraint C16 durch einen Projektor für

$$BMo + BTu + BWe + BTh + BFr \leq 3$$

realisiert werden.

Auch Constraint C15 läßt sich auf ähnliche Weise realisieren. Hier muß noch die Dauer der Kurse bei der Aufsummierung für einen Tag berücksichtigt werden.

Wir wenden uns nun Constraint C9 zu.[2] Dieser Constraint ist disjunktiv. Es muß nämlich für je einen Kurs $C_1$ des einen Lehrers und einen Kurs $C_2$ des anderen

$$start(C_1) + dur(C_1) \leq start(C_2) \ \vee \ start(C_2) + dur(C_2) \leq start(C_1)$$

gelten. Dieser Constraint kann durch Projektoren für die folgenden reifizierten Constraints realisiert werden:

$$\begin{gathered}(start(C_1) + dur(C_1) \leq start(C_2)) \ \leftrightarrow \ B_1 = 1 \\ (start(C_2) + dur(C_2) \leq start(C_1)) \ \leftrightarrow \ B_2 = 1 \\ B_1 + B_2 \geq 1.\end{gathered}$$

Damit ist sichergestellt, daß sich keine zwei Kurse der zwei Lehrer überschneiden. Zusätzlich muß man hier (wie auch bei den folgenden Constraints) noch die notwendigen Pausen nach einem Kurs berücksichtigen, so daß für kurze Kurse noch eine und für lange Kurse noch zwei Viertelstunden zu dem jeweiligen Startzeitpunkt addiert werden müssen.

Ähnlich wie für Constraint C9 kann man auch für C11, C12, C13 und C14 vorgehen.

Für C11 dürfen sich die Vorlesungen eines einzelnen Lehrers paarweise nicht überschneiden.

Für C12 ist es etwas komplizierter, da manche Vorlesungen in verschiedene Kurse unterteilt werden können. Jedoch muß der Stoff jeder Vorlesung von den Studenten eines Semesters gehört werden. Die nicht aufgeteilten Vorlesungen dürfen sich deshalb untereinander nicht überschneiden. Außerdem darf eine nicht aufgeteilte Vorlesung sich mit keiner aufgeteilten Vorlesung überschneiden. Durch explizites Auflisten der Kurse, die sich dann nicht überschneiden dürfen, und durch die Verwendung reifizierter Constraints wie für C9 läßt sich auch Constraint C12 realisieren. Dies gilt auch für Constraint C14.

[2] Dieser Constraint wurde notwendig, da zwei Lehrer ein Kind zu Hause zu betreuen haben. Deshalb können sie nicht beide gleichzeitig unterrichten.

Anders ist es jedoch für aufgeteilte Vorlesungen untereinander (Constraint C13). Wenn es zwischen aufgeteilten Vorlesungen höchstens eine Überschneidung gibt, ist dies eine hinreichende Bedingung dafür, daß jeder Student einen Kurs aus einer aufgeteilten Vorlesung besuchen kann. Sei dazu $C$ ein Kurs und *Courses* eine Menge von Kursen, so daß sich $C$ mit höchstens einem Kurs aus *Courses* überschneiden darf. Für jeden Kurs aus *Courses* reflektiert man den Constraint, daß dieser Kurs sich mit $C$ überschneidet, in eine 0/1-wertige Variable. Die Summe dieser Variablen muß dann kleiner oder gleich eins sein. Dabei überschneiden sich zwei Kurse, wenn für jeden Kurs gilt, daß sein Endzeitpunkt später als der Startzeitpunkt des anderen Kurses ist. Eine Modellierung ist für zwei Kurse $C_1$ und $C_2$ gezeigt.

$$\begin{gathered}(start(C_1) + dur(C_1) > start(C_2)) \leftrightarrow B_1 = 1\\ (start(C_2) + dur(C_2) > start(C_1)) \leftrightarrow B_2 = 1\\ (B_1 + B_2 = 2) \leftrightarrow B = 1\end{gathered}$$

Hierbei ist $B$ gleich 1 genau dann, wenn eine Überschneidung stattfindet.

Es verbleibt nur noch Constraint C10. Dieser Constraint besagt, daß zu keinem Zeitpunkt mehr Kurse stattfinden dürfen, als Räume der erforderlichen Größe vorhanden sind. Ein Kurs $C$ belegt einen Raum zum Zeitpunkt $T$, falls er zwischen $T - dur(C) + 1$ und $T$ beginnt. Diese Information kann durch den reifizierten Constraint

$$(start(C) \in (T - dur(C) + 1 \# T)) \leftrightarrow B = 1$$

in die Variable $B$ reflektiert werden. Zu jedem Zeitpunkt (zwischen 1 und 180) muß für jeden Kurs einer bestimmten Größe eine solche Variable erzeugt werden. Die Summe dieser Variablen darf dann nicht größer sein als die Zahl der verfügbaren Räume. Hierbei ist noch zu beachten, daß Kurse auch in zu großen Räumen stattfinden können.[3]

Damit sind alle Constraints realisiert. Jedoch hat diese Realisierung den Nachteil, sehr viele 0/1-wertige Variablen und reifizierte Constraints zu erzeugen (für eine quantitative Analyse siehe Abschnitt 8.6).

## 8.3 Einsatz globaler Constraints

Um die Anzahl der 0/1-wertigen Variablen und der Projektoren zu reduzieren, können einige Problemconstraints anstatt durch eine große Menge von reifizierten Constraints durch globale Constraints (siehe Abschnitt 4.2) modelliert werden. Kandidaten zu ihrer Anwendung sind die Constraints C9, C10, C11, C12 und C14.

Constraint C11 läßt sich mit dem in Abschnitt 5.2 vorgestellten Kapazitätsconstraint für disjunktive Schedulingprobleme modellieren. Der entsprechende Projektor stellt sicher, daß bei Schedulinganwendungen sich nicht zwei Aufgaben zeitlich in ihrer Ausführung überlappen. Ein Nachteil dieser Realisierung ist jedoch, daß keine Lücken in Bereichen erzeugt werden, die für Propagierung genutzt werden könnten. Deshalb setzen wir hier den Projektor für kumulatives Scheduling ein, da dieser Lücken in Bereichen erzeugt (siehe Abschnitt 5.3 und [HMSW97]).

[3] Damit die Studierenden die Räume wechseln können, muß außerdem für jeden Kurs eine weitere Viertelstunde zur Kursdauer hinzugenommen werden.

Dazu nimmt man einen Ressourcenverbrauch pro Kurs von Eins an. Auch die Kapazität der Ressource Lehrer beträgt Eins.

Auch Constraint C10 kann durch einen Projektor für einen Kapazitätsconstraint für kumulatives Scheduling realisiert werden. Der Realisierung liegt die Einsicht aus Abschnitt 8.1 zugrunde, daß die Räume als Ressourcen mit einer gewissen Kapazität angesehen werden können. Dabei ist der Ressourcenverbrauch eines Kurses für einen Raum entsprechender Größe gleich Eins (da ein Raum belegt wird).

Die Constraints C9, C12 und C14 sind jedoch nicht so einfach mit den in dieser Arbeit vorgestellten Projektoren für globale Constraints zu realisieren. Dies liegt daran, daß hier nur bestimmte Überschneidungen ausgeschlossen sind. Zwar könnte man hierfür spezielle Projektoren zur Verfügung stellen. Doch für die aktuelle Anwendung ist die Effizienz ausreichend, die mit den bisher eingeführten globalen Constraints erreicht wird (siehe Abschnitt 8.7). Eine quantitaive Analyse des Nutzens von globalen Constraints in dieser Anwendung ist in Abschnitt 8.6 zu finden.

## 8.4 Weiche Constraints

Wir haben nun alle Constraints realisiert, die gelten müssen. Dies sind die sogenannten *harten* Constraints. In Abschnitt 8.1 wurden jedoch auch einige *weiche* (engl. *soft*) Constraints über die Verteilung der Kurse aufgeführt, die möglichst gelten sollten. Sie sagen etwas über die Güte des berechneten Stundenplans aus.

Weiche Constraints können durch reifizierte Constraints modelliert werden. Die entsprechende 0/1-wertige Variable erlaubt die Überprüfung, ob ein weicher Constraint erfüllt ist oder nicht. Zum Beispiel kann der weiche Constraint, daß die Vorlesungen des 8. Semesters möglichst dienstags oder donnerstags stattfinden sollen, für einen Kurs $C$ durch

$$(C \in [37\#72\ 109\#144]) \leftrightarrow B = 1$$

modelliert werden.

Da möglichst viele weiche Constraints erfüllt sein sollen, kann man Branch-and-Bound-Suche mit einer geeigneten Bewertungsfunktion anwenden. Diese Bewertungsfunktion muß für eine Lösung die 0/1-wertigen Variablen aufsummieren, die für jeden Kurs durch weiche Constraints eingeführt werden. Außerdem muß sie den Constraint enthalten, daß diese Summe für eine alternative Lösung größer ist als für eine bereits berechnete.

Betrachtet man jedoch die Gütekriterien in Abschnitt 8.1 genauer, so fällt auf, daß sie nicht alle gleichzeitig gut erfüllt werden können. So sollen zum Beispiel die Vorlesungen des zweiten Semesters möglichst montags bis mittwochs plaziert werden. Gleichzeitig sollen aber die Kurse möglichst vormittags stattfinden, was aber für das zweite Semester wegen der Anzahl der Kurse nicht möglich ist. Bei manchen Kursen (wie bei einem des zweiten Semesters am Freitagmorgen) steht der Wunsch des Lehrers sogar mit dem bevorzugten Plazierungsschema im Widerspruch. Deshalb kann man die weichen Constraints gewichten, um ihnen unterschiedliche Wichtigkeit zu verleihen. Damit kann eine Art von *Constrainthierarchie* modelliert werden [WB93].

Sei zum Beispiel *Semester* die Menge der 0/1-wertigen Variablen, die aus den weichen Constraints resultiert, daß ein bestimmtes Semester vorzugsweise an bestimmten Tagen unterrichtet werden soll. Seien *Days* die 0/1-wertigen Variablen, die aus den weichen Constraints herrühren, daß Kurse nicht nachmittags stattfinden sollen. Soll die erste Kategorie weicher Constraints stärker gewichtet werden, so können die Kosten *Costs* einer Lösung zum Beispiel wie folgt berechnet werden.

$$\begin{aligned} \mathit{CostsDays} &= \sum_{d \in \mathit{Days}} d \\ \mathit{CostsSemester} &= \sum_{s \in \mathit{Semester}} s \cdot \mathit{weight}(s) \\ \mathit{Costs} &= \mathit{CostsDays} + \mathit{CostsSemester} \end{aligned}$$

Hierbei ist $\mathit{weight}(s)$ das $s$ entsprechende Gewicht.

Mit diesem Vorgehen lassen sich mäßig gute Resultate erzielen, wenn als Distribuierungsstrategie first-fail verwendet wird. Da für die Wertselektion bei den kleinsten Werten eines Bereichs begonnen wird, werden jedoch automatisch wenig Kurse freitags plaziert.

## 8.5 Distribuierung

In diesem Abschnitt wird eine spezielle Distribuierungsstrategie beschrieben, die indirekt selbst weiche Constraints realisiert. Diese Strategie vermeidet die Verwendung von Constrainthierarchien.

Um möglichst viele weiche Constraints zu erfüllen, gibt man den Werten, die Variablen bei der Distribuierung zugewiesen werden (also bei der Wertselektion), unterschiedliche Priorität. Dazu wird der Bereich 1#180 in 10 Blöcke aufgeteilt, die jeweils einem Vormittag oder Nachmittag eines Tages entsprechen. Diese Blöcke können individuell für jeden Kurs geordnet werden.

Für die vorliegende Anwendung gibt es für jedes Semester eine gemeinsame Ordnung dieser Blöcke, die den Gütekriterien aus Abschnitt 8.1 möglichst nahekommt. Anstatt einer selektierten Variablen den kleinsten Wert ihres Bereichs zuzuweisen, wird der kleinste Wert bezüglich der entsprechenden Block-Ordnung zuerst herangezogen, der in dem Bereich der Variablen noch enthalten ist. Wird also der Block für Dienstagmorgen dem für Montagmorgen vorgezogen, werden erst Werte zwischen 37 und 54, dann Werte zwischen 1 und 18 und dann alle restlichen herangezogen. Dadurch wird erreicht, daß bevorzugte Zeitpunkte zuerst zur Determinierung verwendet werden. Die Variablenauswahl ist wie zuvor durch first-fail gesteuert.

Jedoch finden bei diesem Vorgehen noch relativ viele Kurse nachmittags statt. Dies läßt sich oft auch nicht vermeiden, da manche Lehrer nur nachmittags unterrichten wollen. Aber durch Branch-and-Bound kann der Stundenplan mit einer entsprechenden Bewertungsfunktion, die weiche Constraints bewertet, weiter verbessert werden. Die weichen Constraints besagen gerade, daß ein Kurs möglichst an einem Vormittag stattfinden sollte. Auf diese Weise läßt sich in relativ kurzer Zeit ein verbesserter Plan finden (siehe auch Abschnitt 8.7).

## 8.6 Implementierung

Das in den vorangehenden Abschnitten skizzierte Modell diente als Grundlage für einen Prototyp zur Erstellung von Stundenplänen in DFKI Oz [Pro97]. Ein entsprechendes Programm ist unter [Wür98] zu finden.

Nutzt man keine globalen Constraints, so werden bis zum Erreichen von Stabilität 32 254 FD-Variablen und 31 194 Propagierer (keine Basisconstraints) erzeugt. Dabei sind die meisten FD-Variablen 0/1-wertige Variablen.

Durch den Einsatz globaler Constraints (in diesem Fall von `FD.schedule.cumulative` für kumulatives Scheduling) reduziert sich die Anzahl der FD-Variablen auf 7 234 und die Zahl der Propagierer auf 5 481, also eine Reduktion um etwa den Faktor 6. Von den verbleibenden Variablen und Propagierern gehen etwa 2/3 auf das Konto von Constraint C12 und C13, die nicht von globalen Constraints profitieren.

Mittels einer eigenen Spezifikationssprache kann der Benutzer Bedingungen über die Verfügbarkeit von Lehrern oder Kursen angeben (zum Beispiel bedeutet

```
dayInterval(wednesday # 8#00#12#00)
```

daß der betreffende Lehrer nur mittwochvormittags unterrichten kann). Über ein Menü läßt sich eine entsprechende Datei einlesen. Nach Start der Lösungssuche, wird die erste gefundene Lösung automatisch auf dem Bildschirm angezeigt (siehe Abbildung 8.1). Sofort danach wird automatisch die Branch-and-Bound-Suche gestartet, um eine bessere Lösung zu suchen. Die Suche kann vom Benutzer jederzeit unterbrochen, das Ergebnis inspiziert und evtl. die Suche an der unterbrochenen Stelle wieder aufgenommen werden (*anytime*-Charakteristik). Neben der grafischen Ausgabe steht auch noch eine Ausgabe in Text (Postscript-Datei) oder im Oz-Browser [MMP+97] zur Verfügung. Die Postscript-Datei kann sowohl auf dem Bildschirm als auch auf einem Drucker ausgegegeben werden.

Hilfreich bei der Implementierung waren die grafische Schnittstelle zum Tcl/Tk Toolkit [MS97], die Möglichkeit der objektorientierten Programmierung in Oz und nicht zuletzt die Möglichkeit, Suchstrategien unterbrechen und wieder aufnehmen zu können.

## 8.7 Evaluierung

In diesem Abschnitt werden die unterschiedlichen Lösungsstrategien in der Implementierung aus Abschnitt 8.6 verglichen.

Ein Verzicht auf globale Constraints ist nicht ratsam. Ohne die Verwendung von globalen Constraints wird bei einer einfachen first-fail Strategie etwa 52 MB aktiver Speicher benötigt, um eine erste Lösung zu finden. Bei Verwendung globaler Constraints verringert sich der Speicherverbrauch auf etwa 6 MB (bei vergleichbaren Suchbäumen; siehe unten).

Die Bewertung der verschiedenen Distribuierungsstrategien ist recht schwierig, da die Gütekriterien nicht alle gleichzeitig gut erfüllt werden können. Hier soll die Anzahl der Kurse an einem Nachmittag (außer den freiwilligen Kursen und den Medienkursen – siehe unten) und die An-

zahl der Kurse des 8. Semesters an einem Montag als Kriterien herangezogen werden. Beide Zahlen sollten klein sein. Die Ergebnisse sind in Tabelle 8.1 festgehalten. Es wurden nur 5 000 Wahlpunkte zur Berechnung erlaubt (selbst nach weiteren 5 000 Wahlpunkten wird keine Verbesserung des Zeitplans mehr gefunden). In Tabelle 8.1 enthält die Spalte *first* (bzw. *good*) die Anzahl der Wahlpunkte – in Klammern die Anzahl der Fehlerknoten – im Suchbaum, um die erste Lösung (bzw. beste Lösung im Rahmen der erlaubten 5 000 Wahlpunkte) zu finden. Die Spalte *Mon-first* (bzw. *Mon-good*) enthält die Anzahl der Kurse des 8. Semesters an einem Montag in der ersten (bzw. besten) Lösung und die Spalte *After-first* (bzw. *After-good*) enthält die Anzahl der Kurse am Nachmittag in der ersten (bzw. *besten*) Lösung.

**Tabelle 8.1** Evaluierung der verschiedenen Lösungsstrategien

| Strategie | *first* | *good* | *Mon-first* | *After-first* | *Mon-good* | *After-good* |
|---|---|---|---|---|---|---|
| $D_1$ | 81 (24) | 398 (337) | 6 | 26 | 5 | 25 |
| $D_2$ | 75 (18) | 103 (98) | 7 | 26 | 3 | 25 |
| $D_3$ | 81 (24) | 124 (65) | 6 | 26 | 5 | 25 |
| $D_4$ | 75 (18) | 135 (129) | 7 | 26 | 5 | 20 |
| $D_5$ | 494 (434) | 1 513 (1 451) | 3 | 28 | 2 | 22 |
| $D_6$ | 405 (348) | 632 (627) | 3 | 25 | 2 | 20 |

Die Distribuierungsstrategien $D_1$, $D_3$ und $D_5$ verwenden keine globalen Constraints, während die Strategien $D_2$, $D_4$ und $D_6$ globale Constraints verwenden. $D_1$ und $D_2$ bezeichnen die first-fail Strategie, die in der Bewertungsfunktion eine Constrainthierarchie verwendet (die weichen Constraints sind so gewichtet worden wie in Abschnitt 8.4 beschrieben; es macht keinen Unterschied, ob ein Gewicht von 1, 2 oder 4 verwendet wird). $D_3$ und $D_4$ bezeichnen first-fail mit der Bewertungsfunktion aus Abschnitt 8.5. Die Strategien $D_5$ und $D_6$ bezeichnen die spezielle Distribuierungsstrategie aus Abschnitt 8.5.

Wie man Tabelle 8.1 entnimmt, zahlt sich die Benutzung globaler Constraints durch eine Verkleinerung des Suchbaums und durch bessere Lösungen aus. Die Verkleinerung des Suchbaums kommt zum einen dadurch zustande, daß der verwendete kumulative Propagierer Lücken in Bereichen erzeugt. Die Lücken in den Bereichen werden von allen Distribuierungsstrategien berücksichtigt. Zum anderen resultiert die Verbesserung aus der stärkeren Propagierung der eingesetzten kumulativen Propagierer.

Wie zu erwarten, ist bei Verwendung von first-fail und der üblichen Wertselektion die erste Lösung nach beiden Bewertungskriterien recht schlecht. Durch weitere Suche ergibt sich nur eine schwache Verbesserung bzgl. der Gütekriterien. Während die Anzahl der Nachmittagsvorlesungen durch weitere Suche bei $D_4$ gut reduziert werden kann, ist die Anzahl der Kurse des 8. Semesters am Montag unbefriedigend. Für $D_2$ ist die Qualität der Gütekriterien gerade umgekehrt. Die besten Ergebnisse liefert Strategie $D_6$. Wir werden in Kapitel 12 sehen, daß durch sogenannte konstruktive Disjunktion bessere erste Lösungen gefunden werden können und die Anzahl der Wahlpunkte weiter reduziert werden kann.

Als Alternative verwendeten wir eine spezielle Distribuierungsstrategie für kumulatives Scheduling (siehe [CL96b]). Die Idee hierbei ist es, einen Kurs so früh wie möglich zu plazieren, dabei aber die besonderen Gegebenheiten für Schedulinganwendungen zu berücksichtigen (gelingt eine Plazierung eines Kurses nicht, so können mehr Werte aus dem zugehörigen Bereich entfernt

werden als es eine allgemeine Strategie wie first-fail zulassen würde). Mit dieser Strategie ist nach 10 000 Wahlpunkten noch keine Lösung des Problems gefunden. Inspiziert man den Suchbaum, so fällt auf, daß der Engpaß für dieses Problem offenbar der Mittwochnachmittag ist. Dort müssen nämlich alle Veranstaltungen des sechsten Semesters stattfinden. Eine Strategie für kumulatives Scheduling wird diesen Engpaßbereich jedoch erst sehr tief im Suchbaum erreichen, da der Stundenplan von montagmorgens beginnend aufgefüllt wird. Eine Strategie wie first-fail kann diesen Engpaß viel eher berücksichtigen.

Die erste mit Strategie $D_6$ gefunde Lösung ist in Abbildung 8.1 zu sehen. Die erste Zahl eines Kurseintrags gibt das Semester an (oder `F` für freiwillige Kurse und `M` für Medienkurse). Die zweite Zahl steht für die laufende Nummer der Vorlesung. Eine eventuelle dritte Zahl gibt die Kursnummer an, wenn die Vorlesung in mehrere Kurse aufgeteilt wurde. Es können mehr als sieben Kurse gleichzeitig auf dem Plan erscheinen, da manche Kurse auf das gesamte Semester aufgeteilt werden und somit zwar in der Woche die gleichen Startzeiten haben, aber nur ein halbes (oder drittel) Semester laufen (dies gilt zum Beispiel für `8.15.1` und `8.15.2`, sowie für `2.1`, `2.2` und `2.3`, sowie für einige weitere Kurse). Eine Überschneidung ist für das achte Semester ab der Vorlesung `8.11` erlaubt (wie auch für `6.6` und `6.5`). Ein einzelner Balken rechts an einem Kurseintrag bedeutet, daß dieser Kurs einen mittelgroßen Raum belegt, während zwei Balken anzeigen, daß der Kurs einen großen Raum belegt.

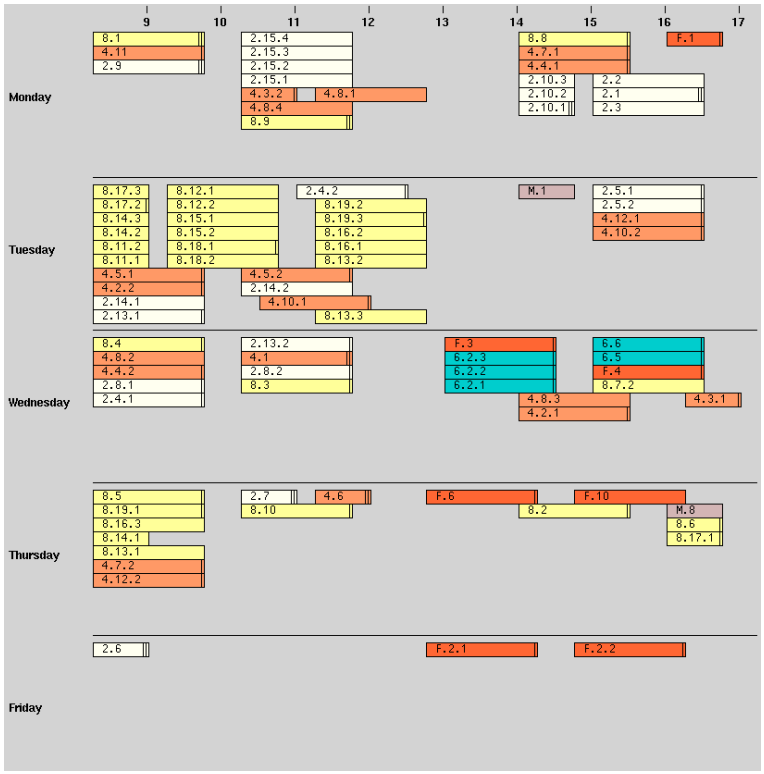
Freiwillige Kurse und Medienkurse dürfen nur nachmittags stattfinden. Sie können somit nicht durch weitere Suche in einen Vormittag verlegt werden. Aufgrund von Constraints und Vorlieben von Lehrern, müssen die Veranstaltungen des 6. Semesters mittwochnachmittags liegen und können deshalb auch nicht in einen Vormittag verschoben werden. Kurse wie `2.10.3` oder `2.1` sind nach der weiterer Suche noch am Montagnachmittag zu finden, da sie relativ weit oben im Suchbaum determiniert werden (im Gegensatz zu Kursen wie `4.7.1`, die dann auch auf einen Vormittag verlegt werden). Somit scheitert ihre Verschiebung an dem nach wie vor großen Suchraum. Berücksichtigt man jedoch, wieviele Kurse überhaupt verlegt werden können, so wird doch fast ein Drittel der Nachmittagskurse durch weitere Suche mit Strategie $D_6$ auf einen Vormittag verlegt.

Mit der Strategie $D_6$ benötigt man zum Finden der ersten (bzw. besten) Lösung 21.8 Sekunden (bzw. 105.4 Sekunden). Verwendet man $D_2$, so benötigt man nur noch 7.9 bzw. 30.2 Sekunden. Die Zeiten wurden auf einer Ultra Sparc 1, 170 MHz, mit dem Betriebssystem SunOS 5.5 gemessen und enthalten nicht die Zeiten für Speicherbereinigung. Es wurde DFKI Oz, Version 2.0.4, verwendet.

## 8.8 Anmerkungen

Stundenplanprobleme sind sehr vielfältig. So werden zum Beispiel Stundenpläne für Gymnasien vor allem durch Raumzwänge beeinflußt, währenddessen für Lehrer und Schüler wenige Constraints vorhanden sind (mit Einschränkungen in der Oberstufe). Auch Probleme zur Planung von Klausurveranstaltungen an Universitäten sind nur durch wenige Constraints eingeschränkt. Dagegen kann es zum Beispiel für Fachhochschulen sehr viele Constraints geben. Einen Überblick über aktuelle Techniken zur Stundenplanerstellung sind in [BR96] zu finden. Einen allgemeinen bibliographischen Überblick gibt [Sch95], wobei jedoch constraintbasierte Ansätze kaum

**Abbildung 8.1** Die erste Lösung mit Strategie $D_6$




Berücksichtigung finden.

In [BDP96] werden Constrainttechniken zur Lösung eines Problems zur Examensplanung vorgestellt. Die dort verwendete Distribuierungsstrategie (*first-fit*) ist aber nicht für das hier beschriebene Problem geeignet, da diese versucht, jedes Semester gleichmäßig auf die Unterrichtswoche zu verteilen und somit die Gütekriterien nur sehr schlecht erfüllt. Auch weitere Suche bringt hier keine zufriedenstellende Lösung.

Einen Überblick über weiche Constraints und verwandte Arbeitsgebiete gibt [MJM96].

# Kapitel 9

# Der Oz-Scheduler

Zur Lösung von Schedulingproblemen gibt es keine generell erfolgreiche Strategie, die Probleme aus unterschiedlichen Bereichen gleich gut löst. Um herauszufinden, auf welche Art Probleme besonders gut und effizient gelöst werden können, müssen oft verschiedene Schedulingtechniken kombiniert und deren Effizienz und Güte getestet werden.

Dies führte zur Realisierung des *Oz-Schedulers*, einer Werkbank zur Lösung von Schedulingproblemen [Wür96, Wür97]. Der Oz-Scheduler ermöglicht das schnelle Testen verschiedener Schedulingtechniken. Dies geschieht mit Hilfe einer grafischen Schnittstelle zu einer vom Benutzer erweiterbaren Schedulingbibliothek. Diese Bibliothek greift zumeist auf die im FD-System von Oz verfügbare Funktionalität zurück (siehe Kapitel 4 und 7 sowie [HMSW97]). In der aktuellen Version des Oz-Schedulers (Ende 1997) wird das Lösen von disjunktiven und kumulativen Schedulingproblemen unterstützt. Wir zeigen, daß heuristische Verfahren wie Iterative Verbesserung mit Constraintprogrammierung in Oz verbunden werden können. Diese Kombination wird im Oz-Scheduler zum raschen Finden guter Lösungen benutzt. Der Oz-Scheduler ist die einzige constraintbasierte Schedulingwerkbank mit einer grafischen Schnittstelle mit der sowohl Optimalitätsbeweise geführt werden können als auch gute Lösungen durch Iterative Verbesserung [CL95, Ree93] gefunden werden können.

Mit dem Oz-Scheduler als einem Tool zum Testen von Schedulingtechniken wird gezeigt, daß die in Oz verfügbaren Techniken kompetitiv zu denen anderer sehr guter Schedulingsysteme sind (wie ILOG SCHEDULER [ILOG96a] oder CLAIRE [CL94b]; siehe Abschnitt 9.4 und 13.3). Außerdem verwenden wir den Oz-Scheduler, um Designentscheidungen im FD-System von Oz zu analysieren. Insbesondere gehen wir hierbei auf Serialisierer ein, die redundante Propagierer zu einem Berechnungsraum hinzufügen können.

In Abschnitt 9.1 werden die Anforderungen an eine Schedulingwerkbank beschrieben. Abschnitt 9.2 beschreibt die Realisierung des Oz-Schedulers. Der folgende Abschnitt 9.3 beschreibt die verwendeten Techniken und in Abschnitt 9.4 werden einige Designentscheidungen in Oz quantitativ analysiert und verschiedene Schedulingtechniken miteinander verglichen.

## 9.1 Anforderungen

Um als Ausgangspunkt für ein umfangreicheres (und evtl. industriell zu nutzendes) Werkzeug zu dienen, muß der Oz-Scheduler rasch und einfach erweitert werden können. Dies betrifft Propagierer und Distribuierer, aber auch neue Problemklassen (zum Beispiel unterbrechbare Aufgaben). Da neue Schedulingtechniken in Oz selbst effizient implementiert werden können, muß der Oz-Scheduler hierfür keine spezielle Funktionalität zur Verfügung stellen. Stattdessen kann das Design des Oz-Schedulers auf die notwendige Infrastruktur für Erweiterungen beschränkt werden.

Bisher wurde zum Finden guter Lösungen von Schedulingproblemen nur Branch-and-Bound-Suche verwendet (siehe zum Beispiel Kapitel 3 und 8). Für Schedulingprobleme ist dies aber nicht immer ausreichend. Bei vielen Problemen ist es einfach zu langwierig, durch Branch-and-Bound zur optimalen Lösung zu gelangen. Oft ist zu Beginn der Suche die verfügbare Information für die Distribuierungsstrategie nicht ausreichend, um wirklich fundierte Entscheidungen treffen zu können und Branch-and-Bound benötigt zu viel Zeit, um diese Entscheidungen durch Rücksetzen (Backtracking) wieder rückgängig zu machen. Es sollte jedoch möglich sein, gute Lösungen schnell zu finden. Andererseits ist man auch an einer Qualitätsgarantie für die bereits gefundenen Lösungen interessiert. Deshalb sollten die bisher betrachteten Suchstrategien durch Strategien zum Finden von oberen und unteren Schranken für die optimale Schedulelänge (das hier betrachtete Optimierungskriterium) von Schedulingproblemen ergänzt werden. Eine Werkbank sollte also idealerweise drei Suchphasen zur Verfügung stellen:

- Eine *Optimierungsphase* zur effizienten Berechnung von guten Lösungen
- Eine *Beweisphase* zur garantierten Berechnung der optimalen Lösung und zum Beweis der Optimalität
- Eine *Bewertungsphase* zur Berechnung von unteren Schranken, um die Qualität von gefundenen Lösungen bewerten zu können

Für jede dieser Phasen sollte der Benutzer verschiedene Techniken kombinieren können, die zu der ausgewählten Problemklasse (bisher disjunktive und kumulative Probleme) passen. Diese beinhalten Distribuierer (wie Serialisierer aus Kapitel 6), Propagierer für Kapazitätsconstraints (siehe Kapitel 5), Bewertungsfunktionen (zum Vergleich von Lösungen) und unterschiedliche Suchstrategien. Die in der Beweis- und Bewertungsphase verwendeten Suchstrategien müssen vollständig sein, um die gewünschten Dienste leisten zu können. Dabei heißt eine Suchstrategie *vollständig*, wenn garantiert alle Lösungen, die einer bestimmen Anforderung genügen, in endlicher Zeit gefunden werden können (zum Beispiel die erste Lösung oder die beste Lösung bzgl. einer bestimmten Bewertungsfunktion).

Die Lösung einer Suchphase sollte inspiziert werden können (evtl. auch grafisch aufbereitet). Zur Bewertung der gewählten Kombination sollten Statistiken über die Lösungssuche zur Verfügung stehen.

Jede Suchphase sollte beliebig unterbrechbar sein, um eine Lösung inspizieren und bewerten zu können. Danach sollte die Suche an der unterbrochenen Stelle fortgesetzt werden können. Damit eignet sich die Beweisphase auch einfach zum Suchen nach einer ersten Lösung.

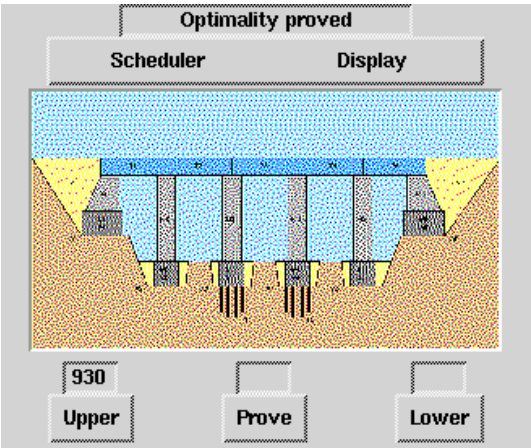
## 9.2 Realisierung

In diesem Abschnitt wird die Realisierung des Oz-Schedulers in DFKI Oz [Pro97] vorgestellt.

### 9.2.1 Die Benutzerschnittstelle

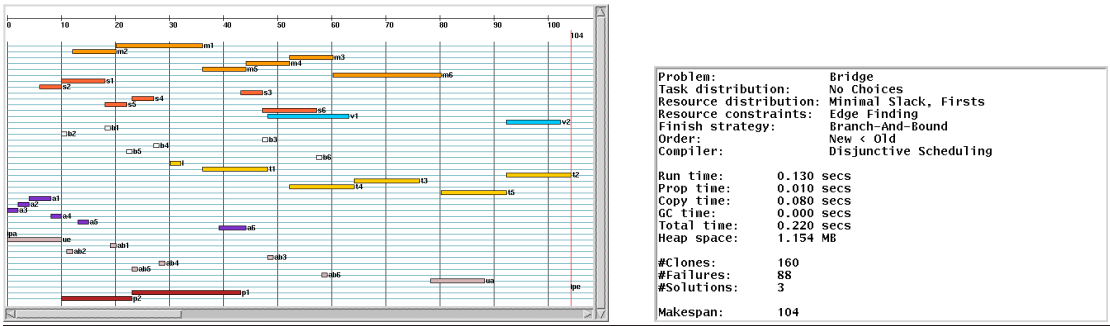
Die grafische Benutzerschnittstelle des Oz-Schedulers ist in Abbildung 9.1 gezeigt. Durch Menüs können die Problemklassen, die zu verwendenden Schedulingtechniken und das zu lösende Problem ausgewählt werden. Für die zur Verfügung stehenden Problemklassen (disjunktive und kumulative Schedulingprobleme) können vordefinierte Kombinationen von Techniken ausgewählt werden.

**Abbildung 9.1** Oz-Scheduler




Eine Lösung kann im Oz-Browser (einem Tool zur Darstellung von (partieller) Information über Variablen [MMP+97]) inspiziert werden. Alternativ gibt es die Möglichkeit, die Lösung eines disjunktiven Schedulingproblems durch ein Gantt-Diagramm grafisch darzustellen (siehe Abbildung 9.2).

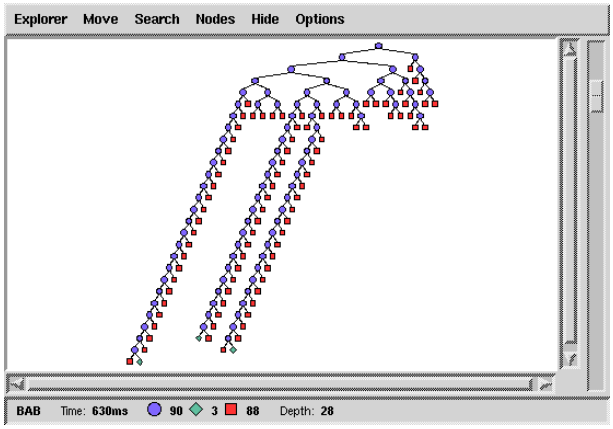
**Abbildung 9.2** Gantt-Diagramm und Statistik



Eine Statistik der zuletzt abgeschlossenen Suchphase kann in einem Fenster auf dem Bildschirm ausgegeben (Abbildung 9.2) oder in einer Datei gespeichert werden. Diese Statistik enthält sowohl Informationen über die verwendete Technikkombination, als auch Angaben zur Laufzeit, zum Speicherverbrauch und zur Lösungssuche.

Zudem ist es möglich, den Suchbaum mit Hilfe des Oz-Explorers (einem Tool zur grafischen Darstellung und Analyse von Suchbäumen [Sch97a, MMP+97]) darzustellen; siehe Abbildung 9.3.

**Abbildung 9.3** Ein Suchbaum im Oz-Explorer




### 9.2.2 Der Aufbau des Oz-Schedulers

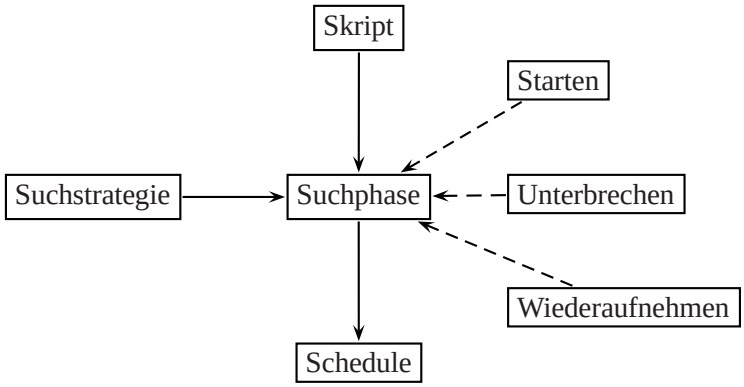
Der Oz-Scheduler ist als ein Objekt implementiert, das durch Instanziierung einer entsprechenden Klasse erzeugt wird (siehe [HSW95, Hen97a] zu objektorientiertem Progammieren in Oz). Die gesamte Funktionalität des Oz-Schedulers steht über Methoden dieser Klasse zur Verfügung. Auch jede der drei Suchphasen ist als Klasse realisiert, die mindestens Methoden zum Starten, Unterbrechen und Wiederaufnehmen der Suche besitzt.

Durch Starten einer Suchphase wird ein Objekt einer entsprechenden Klasse erzeugt. Die Kommunikation zwischen dem Oz-Scheduler und den Suchphasen erfolgt über das Senden von entsprechenden Nachrichten. Es können nicht zwei Suchphasen gleichzeitig laufen (um die Statistik erstellen zu können). Während einer laufenden Suche können jedoch die Techniken für den nächsten Suchlauf ausgewählt werden und die laufende Suchphase kann unterbrochen werden.

Die prinzipielle Funktionsweise einer Suchphase ist in Abbildung 9.4 gezeigt. Die Suchphase ist über die zu verwendende Suchstrategie parametrisiert und kann durch den Benutzer direkt beeinflußt werden (gestrichelte Pfeile).

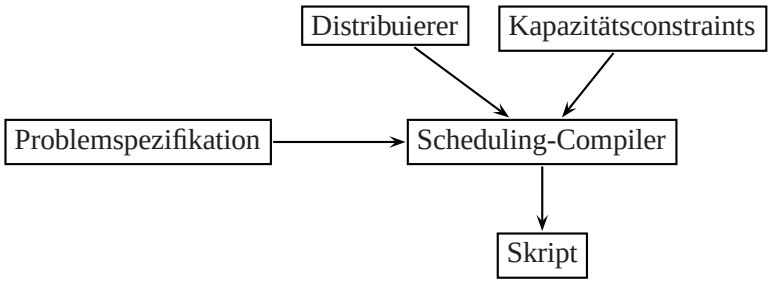
Das Skript, das die Suchphase als Eingabe erhält (siehe Abschnitt 7.1.4), wird durch einen sogenannten *Scheduling-Compiler* erzeugt. Ein Scheduling-Compiler ist eine Oz-Prozedur und ermöglicht die Kombination verschiedener Schedulingtechniken (siehe Abbildung 9.5). Dies geschieht, indem er als Eingabe verschiedene Parameter (wie zum Beispiel die ausgewählte Prozedur zur Realisierung der Kapazitätsconstraints) erhält und als Ausgabe eine Oz-Prozedur (das Skript) erzeugt, die das entsprechende Schedulingproblem modelliert. Sehr hilfreich ist hier, daß in Oz sowohl Prozeduren als auch Klassen über Variablen zur Verfügung stehen. Dies er-

**Abbildung 9.4** Funktionsweise einer Suchphase




leichtert die Erweiterbarkeit, da der Benutzer nur entsprechende Prozeduren bereitstellen muß (siehe unten). Die Problemspezifikation kann sowohl die Repräsentation des Problems als auch noch zusätzliche Constraints enthalten (wie zum Beispiel für das Brückenbauproblem aus Abschnitt 3.2.2). Dies ist vom jeweils verwendeten Scheduling-Compiler abhängig. Die Modularität eines Scheduling-Compilers wird durch die Trennung von Problemrepräsentation, Propagierung und Distribuierung in Oz möglich. Mehr Informationen zu Scheduling-Compilern findet man in [SSW98].

**Abbildung 9.5** Struktur eines Scheduling-Compilers




Ein Scheduling-Compiler für disjunktive Schedulingprobleme ist in dem Oz-Programm 9.1 skizziert. Dabei bezeichnet `Specification` die Problemspezifikation, `CapacityConstraint` die Prozedur, die die Propagierer für Kapazitätsconstraints erzeugt, `Serialization` die Serialisierer und `Assignment` den Distribuierer, der die Startzeiten von Aufgaben determiniert. Der Verbund `StartTimes` enthält die Startzeiten der Aufgaben und ist zusammen mit der Spezifikation Bestandteil des Verbunds, an den die Lösungsvariable des Skripts gebunden wird. Die Aufgaben `Tasks` und die Ausführungszeiten `Durations` werden aus der Spezifikation gewonnen. Der Scheduling-Compiler gibt als Ergebnis das Skript zurück, das die entsprechenden Prozeduraufrufe enthält und das ausgewählte Schedulingproblem modelliert.

Der Scheduling-Compiler für kumulatives Scheduling ist ähnlich aufgebaut. Hier kann aber kein Serialisierer verwendet werden und es müssen andere Kapazitätsconstraints benutzt werden. Auch die Problemrepräsentation ist natürlich verschieden.

**Programm 9.1** Scheduling-Compiler für disjunktive Schedulingprobleme

```
fun {Compile Specification CapacityConstraint Serialization
     Assignment}
   local
      ...
      proc {Script Solution}
         local
            StartTimes
         in
            Solution = r(start: StartTimes
                         spec:  Specification)
            Tasks    = {ExtractTasks Specification}
            Duration = {ExtractDurations Specification}

            ...

            {CapacityConstraint StartTimes Durations Tasks}

            {Serialization StartTimes Durations Tasks}

            {Assignment StartTimes Durations Tasks}
         end
      end
   in
      Script
end
```

Durch das objektorientierte Design des Oz-Schedulers und das Konzept des Scheduling-Compilers ist eine Erweiterung relativ einfach. Neue Propagierer oder Distribuierer können in entsprechenden Nachrichten dem Oz-Scheduler bekannt gemacht werden. Dieser trägt dann diese Techniken in die passenden Menüs ein. Die Verwendung dieser neuen Techniken wird durch die Übergabe von Prozeduren als Parameter des Scheduling-Compilers ermöglicht. Für bestehende Problemklassen müssen die vom Benutzer definierten Techniken eine vorgegebene Signatur einhalten. Neue Problemklassen lassen sich integrieren, indem der Benutzer entsprechende Scheduling-Compiler dem Oz-Scheduler zur Verfügung stellt. Durch neue Scheduling-Compiler ist es zum Beispiel möglich, Problemdaten auf komplexe Weise vorzuverarbeiten oder auf die Problemklasse zugeschnittene Distribuierungsstrategien zu verwenden. Eine neue Suchstrategie für eine Suchphase kann durch eine benutzerdefinierte Klasse integriert werden, die mindestens die Methoden zum Starten, Unterbrechen und Wiederaufnehmen der Suche enthält.

## 9.3 Verwendete Techniken

In diesem Abschnitt wird auf die im Oz-Scheduler verwendeten Lösungstechniken eingegangen. Der Schwerpunkt liegt hierbei auf den Suchphasen.

### 9.3.1 Kapazitätsconstraints

Der Oz-Scheduler bietet für disjunktive und kumulative Schedulingprobleme Propagierer für alle in Kapitel 5 eingeführten Kapazitätsconstraints an. Zusätzlich besteht die Möglichkeit, konstruktive Disjunktion für disjunktive Probleme zu nutzen (siehe Kapitel 12).

### 9.3.2 Distribuierer

Für disjunktive Schedulingprobleme sind die aufgabenorientierten und die ressourcenorientierten Serialisierer aus Kapitel 6 verfügbar. Weiterhin kann der ressourcenorientierte Serialisierer aus Abschnitt 3.2.2 und ein aufgabenorientierter Serialisierer ausgewählt werden, der sich an einer in [SC93] vorgestellten Serialisierung orientiert (die Strategie in [SC93] geht wiederum auf [ERV76, ERV80] zurück).

Für das Determinieren von Startzeiten stehen die üblichen Strategien wie first-fail usw. zur Verfügung. Für kumulatives Scheduling ist eine Strategie verfügbar, die sehr eng mit einer in [CL96b] eingeführten Strategie verwandt ist.

### 9.3.3 Optimierungsphase

Für die Optimierungsphase stehen verschiedene Strategien für sogenannte Job-Shop-Probleme zur Verfügung. Das Optimierungsziel ist die Minimierung der Schedulelänge. Aus der jeweils gefundenen Lösung kann dann eine obere Schranke für die optimale Schedulelänge hergeleitet werden.

Ein *Job-Shop-Problem* [GJ79] besteht aus einer Menge $J$ von Jobs ($|J| = n$). Ein Job $j \in J$ enthält $n_j$ nicht-unterbrechbare Aufgaben $t_i^j, 1 \leq i \leq n_j$, so daß Aufgabe $t_k^j$ vor dem Start von Aufgabe $t_{k+1}^j$ beendet sein muß, $1 \leq k < n_j$. Ein solcher Constraint $start(t_k^j) + dur(t_k^j) \leq start(t_{k+1}^j)$ heißt *Präzedenzconstraint*. Jede Aufgabe $t_k^j$ besitzt eine Ausführungszeit $dur(t_k^j)$ und wird auf einer Maschine $res(t_k^j) \in \{1, \ldots, m\}$ ausgeführt, so daß $res(t_k^j) \neq res(t_{k+1}^j)$ für alle $j \in J$ und $1 \leq k < n_j$ gilt. Alle Aufgaben müssen spätestens zum Zeitpunkt $D \in \mathbb{Z}^+$ fertiggestellt sein. Auf einer Maschine darf zu jedem Zeitpunkt höchstens eine Aufgabe ausgeführt werden. Ein solches Job-Shop-Problem wird auch als $n \times m$-Problem bezeichnet.

Die Suchstrategien der Optimierungsphase bestehen aus zwei Teilen. Im ersten Teil wird eine möglichst gute erste Lösung des Problems durch einen *gierigen Algorithmus* (engl. *greedy algorithm*) konstruiert. Der gierige Algorithmus soll möglichst schnell eine gute Lösung liefern. Im zweiten Teil wird versucht, durch *Iterative Verbesserung* bessere Lösungen zu berechnen. Hierbei wird die Anzahl der erlaubten Fehlerknoten im Suchbaum pro Optimierungsschritt und die Zahl der Schritte selbst begrenzt. Beachte, daß der zweite Teil keine vollständige Suchstrategie implementiert.

Iterative Verbesserung verbindet Constraintprogrammierung mit heuristischen Techniken aus dem Gebiet der lokalen Suche (siehe zum Beispiel [Ree93]). Während der erste Teil der Suchstrategie für Job-Shop-Probleme eingesetzt werden kann, für andere Probleme aus Konstruktionsgründen aber evtl. keine Lösung liefert, kann der zweite Teil für beliebige disjunktive Schedulingprobleme verwendet werden. Die verwendeten Techniken gehen auf [CL95] zurück, vereinfachen aber an manchen Stellen ohne an Effizienz zu verlieren (siehe Abschnitt 13.3). Die gesamte Kontrolle der Optimierungsphase ist in Oz implementiert, greift aber natürlich auf vordefinierte Propagierer und Distribuierer zurück (siehe auch Abschnitt 9.3.6).

**Finden einer guten ersten Lösung** Der *gierige Algorithmus* konstruiert einen Schedule ohne Fehlschlagen durch *Plazierungsregeln mit Prioritäten* (engl. priority dispatching rules, [Fre82, BESW94]). Dabei wird eine Menge $S$ von zu plazierenden Aufgaben verwaltet, die zu Beginn die jeweils erste Aufgabe eines Jobs enthält. Eine Aufgabe $t$ aus $S$ wird nun ausgewählt und so früh wie möglich plaziert. Die übrigen noch nicht plazierten Aufgaben aus $res(t)$ dürfen dann nicht früher als zur Fertigstellungszeit von $t$ beginnen. Anschließend wird $t$ aus $S$ entfernt und der eventuelle Nachfolger von $t$ in seinem Job wird zu $S$ hinzugenommen. Dies wird so lange fortgesetzt, bis alle Aufgaben plaziert sind.

Zur Auswahl der Aufgabe $t$ aus $S$ stellt der Oz-Scheduler Standardtechniken zur Verfügung, wie zum Beispiel diejenige Aufgabe auszuwählen, die die frühest mögliche Startzeit hat (siehe [Law84]). Wir haben außerdem eine Technik aus [CL95] übernommen, die auf [DT93] zurückgeht und sehr gute Ergebnisse liefert.

Für jede Aufgabe $t$ aus $S$ wird unter der Annahme, daß $t$ als nächstes plaziert wird, eine untere Schranke $LB_t$ für die Länge des Schedules wie folgt berechnet (zur Notation siehe Kapitel 5).

$$LB_t = \max\{lb(t') | t' \text{ noch nicht plaziert}\}$$

wobei

$$lb(t) = est(t) + \sum_{res(t')=res(t),\ est(t') \geq est(t)} dur(t').$$

Diejenige Aufgabe $t$ wird als nächstes ausgewählt, für die $LB_t$ minimal ist. Ist dieser Wert für zwei Aufgaben gleich, so wird diejenige Aufgabe mit dem kleinsten Wert von $lst(t)$, der spätest möglichen Startzeit, ausgewählt. Dieses Vorgehen wird durch lokale Berechnungsräume in Oz unterstützt (siehe Abschnitt 9.3.6). Da das Bestimmen von $LB_t$ in $O(n^2 \cdot m)$ möglich ist und dies $O(n \cdot (n \cdot m))$-mal geschieht, hat der Algorithmus eine Zeitkomplexität von $O(n^4 \cdot m^2)$ (im Gegensatz zu $O(n^2 \cdot m)$ der Strategien in [Law84]). Diese hohe Zeitkomplexität wird jedoch durch die gute Qualität der Lösungen ausgewogen (siehe [CL95] und Abschnitt 13.3). Die im Oz-Scheduler verfügbaren Suchstrategien für die Optimierungsphase unterscheiden sich nur in diesem ersten Teil, der zweite Teil der Optimierungsphasen ist gleich.

**Iterative Verbesserung** Bei der Iterativen Verbesserung werden zwei unterschiedliche Techniken verwendet. Die erste versucht, den bisher gefundenen Schedule durch Vertauschen von Aufgaben auf der gleichen Ressource zu verbessern (ein sogenannter *Reparaturschritt*). Die andere Technik versucht mit Hilfe eines sogenannten *Shuffle-Algorithmus* (engl. für Mischen), einen besseren Schedule zu finden, wobei ein Teil der bisherigen Lösung beibehalten wird (zum Beispiel die Serialisierung aller Aufgaben auf einer Ressource). Ein solches Vorgehen ist sinnvoll, da ein partieller Schedule oft durch starke Propagierung (wie durch Edge-Finding) schnell vervollständigt werden kann. Je stärker die Propagierung, desto kleiner sollte der Suchaufwand zur Vervollständigung sein. Die gesamte Struktur der Iterativen Verbesserung ist in Algorithmus 9.1 gezeigt. Der Begriff Iterative Verbesserung kommt daher, daß, von einer bestehenden Lösung ausgehend durch immer weitere Veränderungen eine bessere Lösung gefunden werden soll (ähnlich wie bei Simulated Annealing oder Tabu-Suche; siehe [Ree93, AL97] für einen Überblick über das gesamte Gebiet von Lokaler Suche). Da diese Veränderungen lokal zur bisherigen Lösung sind, wird in [CL95] der Begriff *Lokale Optimierung* verwendet.

**Algorithmus 9.1** Struktur der Iterativen Verbesserung

```
fun itImpr(sol)
   repairSol := repair(sol);
   if better(repairSol, sol)
   then
      return itImpr(repairSol);
   elseif shuffle(shuffleSol) = TRUE
   then
      return itImpr(shuffleSol);
   else
      return sol;
   end;
end;
```

Nur wenn der Reparaturschritt `repair` keine bessere Lösung findet, wird der Shuffle-Algorithmus ausgeführt. Findet auch dieser keine bessere Lösung, wird die bisher beste Lösung als obere Schranke für die optimale Lösung zurückgegeben.

Bei einem *Reparaturschritt* werden sogenannte *kritische Pfade* eines Schedules betrachtet. Ein *Pfad* ist eine Folge von Aufgaben $t_1, \ldots, t_n$, so daß für alle $i \in \{1, \ldots, n-1\}$ gilt, daß $t_i$ direkter Vorgänger von $t_{i+1}$ in einem Job (also $t_i = t_k^j$ und $t_{i+1} = t_{k+1}^j$ für geeignetes $j$ und $k$) oder direkter Vorgänger auf der gleichen Resource ist (also $start(t_i) + dur(t_i) \leq start(t_{i+1})$ und keine Aufga-

be ist zwischen $t_i$ und $t_{i+1}$ auf der gleichen Ressource plaziert). Ein Pfad heißt *kritisch*, wenn seine Länge, also die Summe der Ausführungszeiten der enthaltenen Aufgaben, gleich der Schedulelänge ist. Kommt ein Paar von Aufgaben direkt hintereinander auf allen kritischen Pfaden eines Schedules und auf der gleichen Ressource vor, so kann ein Vertauschen dieser Aufgaben, die Schedulelänge verkürzen. Zu näheren Einzelheiten (zum Beispiel welches Aufgabenpaar ausgewählt wird) siehe [CL95].

Wenn durch den Reparaturschritt keine bessere Lösung gefunden wurde, wird der *Shuffle-Algorithmus* ausgeführt (dieser Begriff wurde in [AC91] geprägt, die Technik geht aber auf [ABZ88] zurück). Der Shuffle-Algorithmus läßt sich in zwei Teile aufspalten, den Basis-Shuffle `basicShuffle` und den Kritischen Shuffle `criticalShuffle`. Die Gesamtstruktur des Algorithmus ist in Algorithmus 9.2 gezeigt.

Der Shuffle-Algorithmus wird durch verschiedene Parameter gesteuert. `UB` ist das Maximum der frühest möglichen Fertigstellungzeiten aller Aufgaben ($\texttt{UB} = \max\{ect(t)\}$), also die aktuell beste obere Schranke für die optimale Schedulelänge. `LB` ist die aktuell beste untere Schranke für die optimale Schedulelänge (entweder Null oder das Ergebnis aus der Bewertungsphase). Weiterhin hat zu Beginn `Iterations` den Wert Zwei, `Leap` und `FailureLimit` den Wert 10 und `MaxFailures` den Wert 90 (siehe unten). Einige der letzteren Parameter definieren eine Art von Ressource, die der Shuffle-Algorithmus noch verbrauchen darf.

Wenn eine verbesserte Lösung gefunden wird, wird diese als Ergebnis des Shuffle-Algorithmus zurückgeliefert. Ansonsten wird der Algorithmus abgebrochen, oder ein neuer Verbesserungsversuch mit veränderten Parametern gemacht. Der Parameter `FailureLimit` bezeichnet die Anzahl der pro Shuffle-Schritt (Basis-Shuffle oder Kritischer Shuffle) erlaubten Fehlerknoten im Suchbaum. Ist diese Anzahl überschritten, wird der jeweilige Shuffle-Schritt abgebrochen. Wird durch Ausführung des Basis-Shuffle und des Kritischen Shuffle keine bessere Lösung gefunden, wird `FailureLimit` erhöht. Dabei wird angenommen, daß die Suche nach einer besseren Lösung nahe am Optimum schwieriger ist und sich ein vermehrter Suchaufwand deshalb lohnt. Wenn `FailureLimit` den Wert `MaxFailures` erreicht, wird der Shuffle-Algorithmus abgebrochen. Der Parameter `Leap` bezeichnet den Wert, um den eine alternative Lösung besser sein soll als die bisherige. Ist die aktuelle obere Schranke nahe an der aktuellen unteren Schranke wird `Leap` verringert. Dies soll sicherstellen, daß die optimale Lösung nicht verfehlt wird. `Iterations` bezeichnet die Anzahl der erlaubten Durchgänge des Shuffle-Algorithmus, wenn `Leap` den Wert Eins angenommen hat und die Zahl der erlaubten Fehlerknoten `MaxFailures` noch nicht erreicht ist.

Der Basis-Shuffle `basicShuffle` extrahiert aus einem bestehenden Schedule die Serialisierung für eine Ressource *r*. Das Skript (siehe Abschnitt 7.1.4) für den nachfolgenden Schritt besteht aus dem Aufruf des durch den Scheduling-Compiler erzeugten Skripts, aus zusätzlichen Ordnungsconstraints der Serialisierung von *r* und dem Constraint, daß die alternative Lösung höchstens eine Länge von `UB-Leap` haben darf (also um mindestens `Leap` besser sein muß). Die Serialisierung der von *r* verschiedenen Ressourcen wird nicht übernommen. Mit den vom Benutzer ausgewählten Schedulingtechniken wird versucht, eine Lösung zu finden. Wird eine Lösung gefunden, so wird diese zurückgegegeben. Ansonsten wird eine nächste Ressource $r'$ ausgewählt und wie für *r* verfahren. Wird für keine Ressource ein besserer Schedule gefunden, wird die bisherige Lösung zurückgegeben. Es wird jeweils diejenige Ressource zuerst ausgewählt, die den größten lokalen Schlupf hat (siehe Kapitel 6, Seite 93). Damit wird die Ressource zuerst

**Algorithmus 9.2** Shuffle-Algorithmus

```
fun shuffle(sol)
   if UB-Leap < LB
   then if Leap > 1
        then Leap := Leap div 3;
           return TRUE;
        else return FALSE;
        end;
   else
      basicSol := basicShuffle(sol);
      if better(basicSol, sol)
      then sol := basicSol;
         return TRUE;
      else
         criticalSol := criticalShuffle(sol);
         if better(criticalSol, sol)
         then sol := criticalSol;
            return TRUE;
         else
            if Leap > 1
            then if FailureLimit >= MaxFailures
               then return FALSE;
               else Leap := Leap div 3;
                  FailureLimit := FailureLimit * 3;
                  return shuffle(sol);
               end;
            else
               if Iterations > 0
               then if FailureLimit >= MaxFailures
                  then return FALSE;
                  else
                     FailureLimit := FailureLimit * 3;
                     Iterations := Iterations - 1;
                     return shuffle(sol);
                  end;
               end;
            end;
         end;
      end;
   end;
end;
```

fixiert, für die am meisten Spiel für die Verschiebung von Aufgaben bleibt.

Für den Kritischen Shuffle `criticalShuffle` sei $R$ die Liste der Ressourcen nach kleiner werdendem lokalen Schlupf geordnet. Nacheinander werden zwei aufeinanderfolgende Ressourcen $r_1$ und $r_2$ aus $R$ ausgewählt. Für $r_1$ und $r_2$ werden alle Aufgabenpaare $(t,t')$ bestimmt, für die $t$ und $t'$ auf der gleichen Ressource ausgeführt werden und $t'$ direkter Nachfolger von $t$ ist. Kommt für ein Paar $(t,t')$ die Aufgabe $t'$ auf einem kritischen Pfad vor, so wird das vom Scheduling-Compiler erzeugte Skript durch den Constraint $start(t) + dur(t) \leq start(t')$ ergänzt. Für die übrigen Aufgabenpaare ist eine Ordnung durch Suche zu finden. Wie bei dem Basis-Shuffle wird zusätzlich der Constraint hinzugefügt, daß die alternative Lösung höchstens eine Länge von `UB-Leap` haben darf. Wird eine Lösung gefunden, so wird diese zurückgegegeben. Ansonsten wird das nächste Ressourcenpaar herangezogen. Wird für keine Ressource ein besserer Schedule gefunden, wird die bisherige Lösung zurückgegeben.

Die Unvollständigkeit des Shuffle-Algorithmus stammt also aus der Begrenzung der erlaubten Fehlerknoten pro Schritt, aus der begrenzten Zahl von Problemvarianten, die aus einer bereits gefundenen Lösung gewonnen werden und aus der begrenzten Anzahl Durchläufe durch den Shuffle-Algorithmus selbst.

In [CL95] wird ein weiterer Shuffle-Algorithmus und eine weitere Variante des Reparaturalgorithmus verwendet. Es zeigt sich jedoch, daß die hier aufgeführte einfache Variante in Verbindung mit den zur Verfügung stehenden Kapazitätsconstraints und Distribuierungsstrategien meist ausreichend ist (siehe Abschnitt 9.4 und 13.3).

### 9.3.4 Beweisphase

Für die Beweisphase steht (Ende 1997) nur die Branch-and-Bound-Suchstrategie zur Verfügung. Die entsprechende Bewertungsfunktion enthält den Constraint, daß die Länge des alternativen Schedules echt kleiner sein muß als die Länge des bereits gefundenen Schedules. Das Skript für diese Phase besteht aus dem vom Scheduling-Compiler erzeugten Skript, zu dem der Constraint hinzugefügt wird, daß die Länge des Schedules kleiner als die beste verfügbare obere Schranke aus der Optimierungsphase bzw. die Summe der Ausführungszeiten der Aufgaben plus Eins sein muß. Es können beliebige vollständige Suchstrategien für diese Suchphase verwendet werden. Beachte, daß für ein Job-Shop-Problem eine Serialisierung der Aufgaben zur Lösung des Problems ausreicht. Die Lösung mit der kleinsten Schedulelänge für diese Serialisierung kann dann ohne Suche dadurch bestimmt werden, daß der Startzeitpunkt jeder Aufgabe zu dem kleinsten Wert in dem entsprechenden Bereich determiniert wird. Dies folgt unmittelbar aus der Tatsache, daß es nur Präzedenzconstraints der Form $x + c \leq y$ ($c$ ist eine ganze Zahl) und Kapazitätsconstraints gibt und nach der Serialisierung für jedes Paar von Startzeitpunkten $(x,y)$ entweder $x + c_1 \leq y$ oder $y + c_2 \leq x$ als Propagierer vorhanden oder bereits subsumiert ist. Damit ist aber offensichtlich, daß nach Erreichen von Stabilität eine Lösung durch Determinieren jeder Variablen auf ihren kleinsten noch möglichen Wert gewonnen werden kann (siehe auch [VD91a] und Abschnitt 10.3.2).

### 9.3.5 Bewertungsphase

Wie für die Beweisphase muß in der Bewertungsphase eine vollständige Suchstrategie verwendet werden, da bewiesen werden muß, daß es für eine bestimmte Schedulelänge keine Lösung gibt; der entstandene Suchbaum also nicht weiter expandiert werden kann. Um sukzessive bessere untere Schranken zu erhalten, wurde eine binäre Suche über die Schedulelänge implementiert. Zu Beginn wird für die Länge ein Wert aus dem Intervall zwischen Null und der Summe der Ausführungszeiten der Aufgaben bzw. der besten verfügbaren oberen Schranke aus der Optimierungsphase angenommen. Bezeichne $[l,u]$ das aktuelle Intervall. Eine untere Schranke für die Länge eines Schedules ist die linke Grenze $l$ dieses Intervalls. Dann wird mit Tiefensuche nach der ersten Lösung des Skripts gesucht, das aus dem vom Scheduling-Compiler erzeugten Skript durch Hinzufügen des Constraints entsteht, daß die Länge des Schedules kleiner oder gleich dem Wert $m = \lfloor (l+u)/2 \rfloor$ sein muß ($\lfloor x \rfloor$ bezeichnet die größte ganze Zahl, die kleiner oder gleich $x$ ist). Wird für dieses Problem eine Lösung gefunden, wird das Intervall zu $[l,m]$ verkleinert. Ansonsten wird das Intervall zu $[m,u]$ reduziert. Die Verkleinerung des Intervalls wird so lange fortgesetzt, bis $l+1 = u$ gilt oder die Suche unterbrochen wird. Im ersten Fall ist $u$ die Schedulelänge der optimalen Lösung.

Eine untere Schranke kann man auch erhalten, indem man zu dem vom Scheduling-Compiler erzeugten Skript den Constraint hinzufügt, daß die Schedulelänge kleiner als ein bestimmter Wert sein soll. Führt dies allein durch Propagierung zum Widerspruch, so ist dieser Wert eine untere Schranke. Auf diese Weise kann oft schnell eine untere Schranke bestimmt werden. Alternativ kann man auch einen Schedule nur für eine Maschine berechnen und dessen Länge als untere Grenze verwenden (siehe zum Beispiel [Kan76, CL95]). In [AC91] werden sogenannte Schnittebenentechniken zur Bestimmung unterer Schranken verwendet. Siehe Abschnitt 13.3 für einen quantitativen Vergleich.

### 9.3.6 Implementierungsaspekte

Die Möglichkeit, daß Suchphasen unterbrochen und die Suche später wieder aufgenommen werden kann, wird von entsprechender Funktionalität der vordefinierten Suchstrategien in Oz unterstützt. Die Bewertungsphase profitiert von der Möglichkeit, in Oz neue Skripten durch die Integration von bestehenden Skripten zu definieren (um die Schedulelänge zu begrenzen). Gleiches gilt auch für die Optimierungsphase, wo bestehende Skripten z. B. durch Ordnungsconstraints für Aufgaben auf einer Ressource ergänzt werden. Auch für den gierigen Algorithmus werden bestehende Skripte durch weitere Constraints ergänzt. Durch das Konzept der Berechnungsräume und deren Funktionalität ist es möglich, auf die Propagierung einer Aufgabenplazierung zu synchronisieren und das Ergebnis zu reflektieren. Für die Shuffle-Schritte ist es notwendig, eine Suchstrategie zu implementieren, die über die Anzahl der erlaubten Fehlerknoten parametrisiert ist. Die Implementierung einer solchen Suchstrategie wird wiederum durch die Existenz von Berechnungsräumen und deren Funktionalität in Oz unterstützt.

## 9.4 Evaluierung der Schedulingtechniken in Oz

In Kapitel 5 und 6 wurden die Unterschiede in der Laufzeitkomplexität und in der Stärke der Propagierung von Kapazitätsconstraints und Serialisierern aufgezeigt. In diesem Abschnitt wird anhand von Beispielen gezeigt, welche Auswirkungen diese Unterschiede auf das konkrete Laufzeitverhalten haben (geht doch stärkere Propagierung und damit eine Reduktion des Suchbaumes oft mit einer höheren Laufzeitkomplexität einher). Hierzu werden die verschiedenen Techniken mit Hilfe des Oz-Schedulers bewertet. Außerdem werden einige Designentscheidungen in Oz für die in Kapitel 5 und 6 vorgestellten Kapazitätsconstraints und Serialisierer untersucht, die die spezielle Kombination von Schedulingtechniken und Constraintprogrammierung betreffen. Alle Messungen wurden auf einer Sun Ultra-1, 170 MHz, 320 MB Hauptspeicher, Betriebssystem SunOS 5.5 ausgeführt. Die Laufzeit beinhaltet auch die Zeit für Speicherbereinigung und kopierbasierter Suche (siehe [MMP+97]). Es wurde die Standardeinstellung der Parameter für die Speicherbereinigung verwendet. Alle verwendeten Programme sind elektronisch verfügbar unter [Wür98].

Wir betrachten hier die folgenden zehn Job-Shop-Probleme mit je 10 Jobs á 10 Aufgaben, die auf 10 Maschinen ausgeführt werden. Das Problem MT10 stammt aus [MT63], die Probleme ABZ5 und ABZ6 aus [ABZ88], die Probleme LA19 und LA20 aus [Law84] und ORB1 bis ORB5 aus [AC91]. Diese Probleme sind klassische Probleme zum Effizienztest für Algorithmen aus dem Operations Research und gelten als schwierig. So wurde zum Beispiel ein Optimalitätsbeweis für das Problem MT10 erst 1989 [CP89], also mehr als 25 Jahre nach der Problemdefinition geführt.

### 9.4.1 Evaluierung verschiedener Schedulingtechniken

**Optimalitätsbeweis** In diesem Abschnitt werden verschiedene in Oz implementierte Schedulingtechniken für den Optimalitätsbeweis verglichen. Dies bedeutet, daß das vom Schedulingcompiler erzeugte Skript durch Constraints ergänzt wird, daß der Fertigstellungszeitpunkt jeder Aufgabe echt kleiner sein muß als die Schedulelänge der optimalen Lösung.

Auf der horizontalen Achse von Abbildung 9.6 sind die Job-Shop-Probleme in der Reihenfolge MT10, ABZ5, ABZ6, LA19, LA20, ORB1, ORB2, ORB3, ORB4, ORB5 aufgetragen. Die vertikale Achse gibt die CPU-Zeit in Sekunden wieder, die für den Optimalitätsbeweis benötigt wird. Werden mehr als 500 Sekunden benötigt, so wird dem entsprechenden Symbol für die Technikkombination ein Pfeil $\uparrow$ vorangestellt. In Abbildung 9.7 ist auf der vertikalen Achse die Anzahl der für einen Optimalitätsbeweis notwendigen Fehlerknoten im Suchbaum im logarithmischen Maßstab aufgetragen.

Tabelle 9.1 definiert die verwendeten Kombinationen aus Kapazitätsconstraints und Serialisierern durch die entprechenden Prozeduren im FD-System von Oz (siehe [HMSW97]). Die Kombinationen $\bullet$, $\triangle$, $\diamond$ und $\triangleleft$ verwenden den aufgabenorientierten Serialisierer aus Abschnitt 6.3 (die Variante speziell für den Optimalitätsbeweis), der in DFKI Oz über `FD.schedule.taskIntervalsDistP` zur Verfügung steht. Die übrigen Kombinationen verwenden die ressourcenorientierten Serialisierer aus Abschnitt 6.2. In Kombination $\ominus$ werden nur diejenigen Aufgaben als Kandidaten für die Serialisierung betrachtet, die zuerst auf einer Ressource plaziert werden können, also `FD.schedule.firstsDist` in DFKI Oz. Für $\oplus$ sind es die Aufgaben, die zuletzt (also `FD.schedule.lastsDist` in DFKI

**Tabelle 9.1** Symbolerklärungen für die verwendeten Kombinationen

| Symbol | Kapazitätsconstraint | Serialisierer |
|---|---|---|
| $\bullet$ | `FD.schedule.serialized` | `FD.schedule.taskIntervalsDistP` |
| $\triangle$ | `FD.schedule.taskIntervals` | `FD.schedule.taskIntervalsDistP` |
| $\diamond$ | `FD.schedule.disjunctive` | `FD.schedule.taskIntervalsDistP` |
| $\triangleleft$ | `FD.schedule.cumulative` | `FD.schedule.taskIntervalsDistP` |
| $\ominus$ | `FD.schedule.serialized` | `FD.schedule.firstsDist` |
| $\oplus$ | `FD.schedule.serialized` | `FD.schedule.lastsDist` |
| $\otimes$ | `FD.schedule.serialized` | `FD.schedule.firstsLastsDist` |
| $\odot$ | `FD.schedule.taskIntervals` | `FD.schedule.firstsLastsDist` |

Oz), und für $\otimes$ die Aufgaben, die zuerst oder zuletzt plaziert werden können (siehe Seite 95), also `FD.schedule.firstsLastsDist`. Die Kombinationen $\ominus$, $\oplus$, $\otimes$ und $\bullet$ verwenden alle den Kapazitätsconstraint aus Abschnitt 5.2.4 (den Zwei-Phasen-Algorithmus), der in DFKI Oz über `FD.schedule.serialized` verfügbar ist. Kombination $\triangle$ und $\odot$ verwenden Aufgabenintervalle (siehe Abschnitt 5.2.3); der entsprechende Propagierer ist über `FD.schedule.taskIntervals` verfügbar. Kombination $\diamond$ verwendet den globalen Constraint aus Abschnitt 5.2.1 (über `FD.schedule.disjunctive` verfügbar) und $\triangleleft$ den kumulativen Kapazitätsconstraint für den Fall des disjunktiven Schedulings aus Abschnitt 5.3 (über `FD.schedule.cumulative` verfügbar).

Offensichtlich ist die aufgabenorientierte Strategie `FD.schedule.taskIntervalsDistP` der beste Distribuierer in Bezug auf die notwendigen Fehlerknoten. Fast immer benötigen die Kombinationen mit den ressourcenorientierten Serialisierern mehr Fehlerknoten. Dies gilt auch, wenn man wie für Kombination $\odot$ Aufgabenintervalle für den Kapazitätsconstraint verwendet. In diesem Fall erzielt man auch noch die schlechtesten Laufzeiten. Verwendet man jeweils `FD.schedule.serialized` als Kapazitätsconstraint, wird der Komplexitätsunterschied der Serialisierer in der Laufzeit offenbar. So sind die ressourcenorientierten Serialisierer effizienter als der aufgabenorientierte Serialisierer (pro Sekunde werden für Probleme dieser Größenordnung etwa zweimal mehr Fehlerknoten besucht). Dies ist vielleicht etwas überraschend, da der Distribuierer doch nur bei Stabilität aufgerufen wird. Jedoch hat der ressourcenorientierte Serialisierer eine lineare Komplexität in der Zahl der Aufgaben pro Ressource (wenn keine neue Ressource zur Serialisierung ausgewählt wird), der aufgabenorientierte Serialisierer hingegen immer eine kubische Komplexität in der Zahl der Aufgaben insgesamt (siehe auch Kapitel 6). Somit sind die ressourcenorientierten Serialisierer gute Kandidaten für Schedulingprobleme mit vielen Aufgaben (siehe auch Kapitel 10).

Durch die Verwendung von `FD.schedule.taskIntervals` als Kapazitätsconstraint läßt sich die Anzahl der Fehlerknoten im Vergleich zur Verwendung des Zwei-Phasen-Algorithmus reduzieren. Jedoch macht sich die Laufzeitkomplexität des Kapazitätsconstraints mit Aufgabenintervallen in Oz so stark bemerkbar, daß in der Regel die Kombination mit dem Kapazitätsconstraint mit dem Zwei-Phasen-Algorithmus weitaus schneller ist.

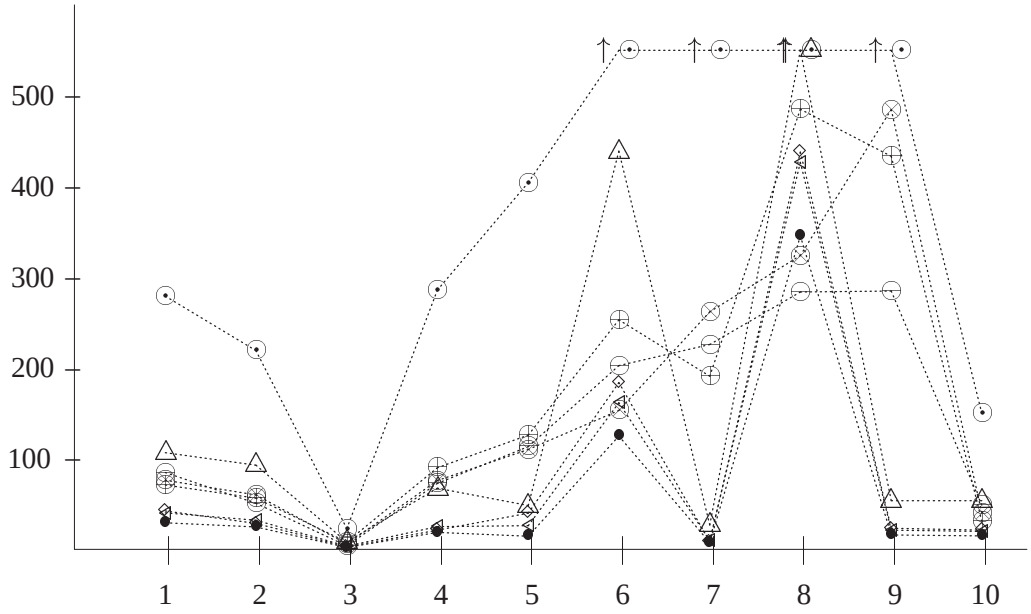
Der kumulative Kapazitätsconstraint liefert nur leicht schlechtere Ergebnisse als das stärker propagierende Edge-Finding von `FD.schedule.serialized`. Dies wird plausibel, schaut man sich die Kombination $\diamond$ mit `FD.schedule.disjunctive` als Kapazitätsconstraint an. Obwohl

hier nur sehr wenig Propagierung durch den Kapazitätsconstraint stattfindet, werden doch nur bis zu doppelt so viele Fehlerknoten benötigt. Dies liegt an dem sehr guten Distribuierer, der ja auch gleichzeitig durch Edge-Finding redundante Propagierer zum Berechnungsraum hinzufügt (siehe unten). Da der Kapazitätsconstraint jedoch weniger Berechnung durchzuführen hat als `FD.schedule.serialized`, ist die resultierende Laufzeit recht gut.

Verwendet man für den Optimalitätsbeweis für MT10 den naiven Serialisierer aus Abschnitt 3.2 für das Brückenbauproblem und den Zwei-Phasen-Algorithmus, so benötigt man 41 835 Fehlerknoten. Dieses noch relativ gute Abschneiden liegt an der starken Propagierung durch Edge-Finding (siehe aber auch den folgenden Paragraphen). Der Optimalitätsbeweis für das Brückenbauproblem kann mit Kombination $\otimes$ mit 5 Fehlerknoten erbracht werden. Die beste Variante in Kapitel 3 benötigt hierfür 230 Fehlerknoten.

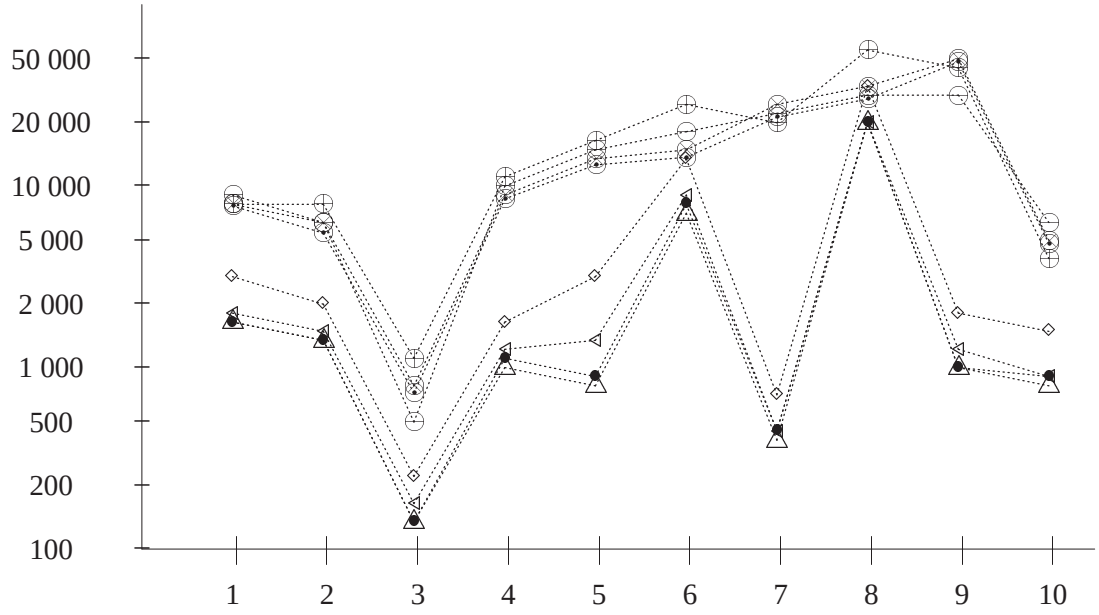
Die Kombination des Serialisierers `FD.schedule.taskIntervalsDistP` mit dem Kapazitätsconstraint `FD.schedule.serialized` führt also zu den besten Ergebnissen. Zu einer genaueren Laufzeitanalyse siehe Abschnitt 9.4.3.

**Abbildung 9.6** Laufzeit für einen Optimalitätsbeweis




**Uninformierte Suche** Der Serialisierer `FD.schedule.taskIntervalsDistP` schneidet beim Optimalitätsbeweis auch deshalb so gut ab, da hier die Bereiche der Startzeitvariablen schon relativ stark eingeschränkt sind (der Schlupf einer Variablen ist also recht klein). Damit ist gewährleistet, daß die Ordnungsentscheidungen im oberen Teil des Suchbaumes schon gut fundiert sind. Startet man aber eine Branch-and-Bound-Suche zum Beispiel für MT10 ohne anfängliche Einschränkung der Schedulelänge, benötigt man mit der aufgabenorientierten Strategie `FD.schedule.taskIntervalsDistO` (dies ist die Strategie, die für das Finden von (guten) Lösungen spezialisiert ist; siehe Seite 101) und dem Zwei-Phasen-Algorithmus 19 948 Fehlerknoten zum Finden und Beweisen der optimalen Lösung. Verwendet man hingegegn

**Abbildung 9.7** Anzahl der Fehlerknoten für einen Optimalitätsbeweis




`FD.schedule.firstsLastsDist` als Serialisierer, so benötigt man nur 16 696 Fehlerknoten. Interessanter ist aber die Qualität der gefundenen Lösungen. So besitzt die erste Lösung, die mit dem aufgabenorientierten Serialiserer (mit 66 Fehlerknoten) gefunden wird, die Länge 2 523, während die mit Kombination $\otimes$ (mit Null Fehlerknoten) gefundene Schedulelänge nur 1 232 beträgt. Dies macht deutlich, daß bei weniger Information über die Startzeiten eine ressourcenorientierte Strategie gute Resultate erzielen kann. Die Entscheidungen des aufgabenorientierten Serialisierers lenken in diesem Fall die Suche früh in eine (möglicherweise) ungünstige Richtung.

Ähnliches gilt auch für Problem MT10 bei Verwendung des für das Brückenbauproblem definierten Serialisierers und des Zwei-Phasen-Algorithmus. Für das Finden und Beweisen der optimalen Lösung werden dann 1 533 690 Fehlerknoten benötigt. Da Edge-Finding nun zu Beginn der Suche wenig Propagierung beitragen kann, wird die schlechte Qualität des Serialisierers besonders deutlich. Um die optimale Lösung des Brückenbauproblems zu finden und ihre Optimalität zu beweisen benötigt Kombination $\otimes$ nur 36 Fehlerknoten. Die beste Variante in Kapitel 3 benötigt hierfür 434 Fehlerknoten.

**Tiefe von Suchbäumen** Wichtig ist es auch, die Tiefe der entstehenden Suchbäume zu betrachten, da dies den Speicherverbrauch direkt beeinflußt. Theoretisch ist für den aufgabenorientierten Serialisierer `FD.schedule.taskIntervalsDistP` eine große Suchtiefe zu erwarten. Jedoch besitzt der Suchbaum für den Optimalitätsbeweis von MT10 mit dem Zwei-Phasen-Algorithmus nur eine Tiefe von 39. Der Suchbaum mit der ressourcenorientierten Strategie `FD.schedule.firstsLastsDist` hat eine Tiefe von 36. Dies wird jedoch plausibel, wenn man bedenkt, daß der aufgabenorientierte Serialisierer viele durch Propagierung determinierte Ordnungsentscheidungen erkennen kann und durch Hinzufügen redundanter Propagierer zum Berechnungsraum realisiert, so daß hierfür keine Wahlpunkte angelegt werden müssen (siehe

Abschnitt 7.3.4). Wichtig ist jedoch auch, daß starke Propagierung durch Edge-Finding (gerade auch des Serialisieres) dafür sorgt, daß früh Teile des Suchraumes von einer Explorierung ausgeschlossen werden können. Ist man jedoch an der ersten Lösung interessiert (siehe oben), hat der Suchbaum mit `FD.schedule.taskIntervalsDistO` eine Tiefe von 451 und der mit der Kombination $\otimes$ nur eine Tiefe von 91. Da nun die Bereiche der Startzeiten weniger eingeschränkt sind, können kaum Ordnungsentscheidungen determiniert werden. Zudem kann Edge-Finding nicht viel Propagierung beitragen, so daß der Suchraum nur wenig reduziert wird. Dies ist ein deutlicher Hinweis, daß bei großen nicht stark eingeschränkten Problemen eine ressourcenorientierte Strategie eine gute Wahl sein kann (siehe auch Abschnitt 10).

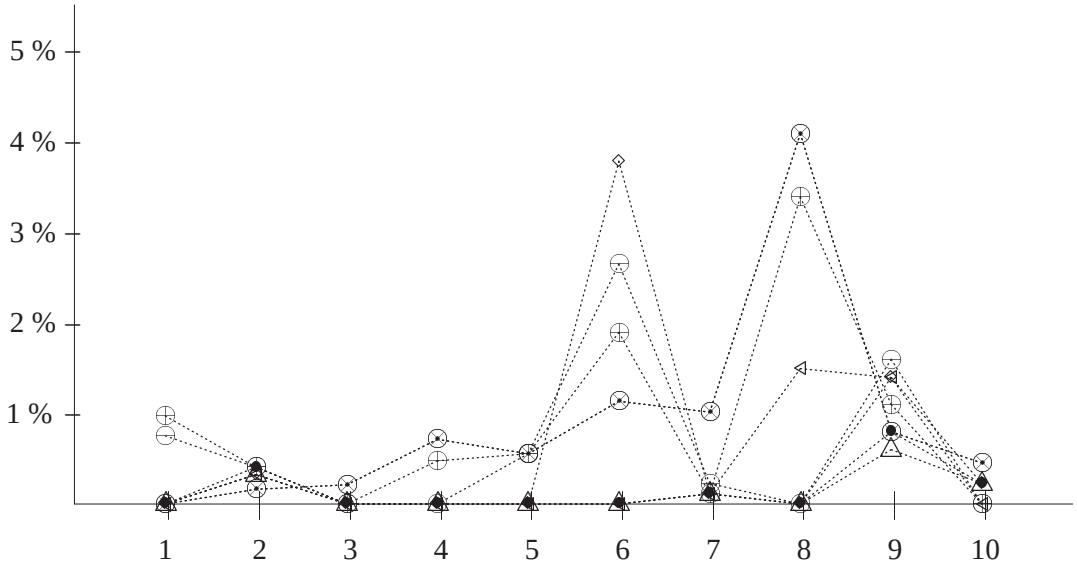
**Optimierung** Abbildung 9.8 enthält die Ergebnisse für die Optimierungsphase mit den Parametern aus Abschnitt 9.3.3 und den Kombinationen aus Tabelle 9.1, wobei nun jedoch `FD.schedule.taskIntervalsDistO` als aufgabenorientierter Serialisierer verwendet wird. Auf der vertikalen Achse ist die Abweichung der besten gefundenen Schedulelänge von dem optimalen Wert angegeben (also die Differenz aus der gefundenen Schedulelänge und dem Optimum geteilt durch das Optimum). Die Anzahl der benötigten Fehlerknoten für verschiedene Kombinationen ist in der Regel für ein Problem sehr ähnlich. Auch die Laufzeiten pro Problem sind nicht signifikant verschieden (mit Ausnahme der Kombination $\triangle$ und $\odot$, für die berechnungsintensive Aufgabenintervalle als Kapazitätsconstraints verwendet werden). Die optimalen Schedulelängen sind in Tabelle 9.2 in der Spalte *Optimum* enthalten.

**Tabelle 9.2** Optimale Schedulelängen

| Problem | Optimum |
|---|---|
| MT10 | 930 |
| ABZ5 | 1234 |
| ABZ6 | 943 |
| LA19 | 842 |
| LA20 | 902 |
| ORB1 | 1059 |
| ORB2 | 888 |
| ORB3 | 1005 |
| ORB4 | 1005 |
| ORB5 | 887 |

Analysiert man Abbildung 9.8, so schneidet die Kombination $\triangle$ am besten ab, was die Qualität der Lösung betrifft. Wie für den Optimalitätsbeweis führen Aufgabenintervalle aber auch hier zu den längsten Laufzeiten. Berücksichtigt man Qualität und Effizienz, so erzielt die Kombination $\bullet$ die besten Ergebnisse. In den meisten Fällen liefern die Kombinationen, die ressourcenorientierte Serialisierer verwenden, nicht so gute Ergebnisse. Die Verwendung von Aufgabenintervallen in Kombination $\odot$ ergibt genau die gleichen Schedulelängen wie für $\otimes$, jedoch steigt die Laufzeit auf mehr als das Zweifache an. Das gute Abschneiden des aufgabenorientierten Serialisierers steht nicht im Widerspruch zur schlechten Qualität einer ersten Lösung, wenn die Schedulelänge gar nicht eingeschränkt wird (siehe oben). Zum einen ist in der Optimierungsphase die Schedulelänge aufgrund eines guten Startwertes aus dem gierigen Algorithmus schon recht klein (siehe Abschnitt 13.3). Zum anderen wird bei einem Shuffle-Schritt zum Beispiel die Serialisierung

**Abbildung 9.8** Abweichung der besten gefundenen Lösung vom Optimum




einer Ressource komplett übernommen. Zusammen führt dies zu starker Reduktion vieler Bereiche, so daß die Entscheidungen des Serialisierers gut fundiert sind.

Beachte, daß mit der Kombination • jedoch alle optimalen Schedulelängen gefunden werden können, wenn die Parameter der Optimierungsphase erhöht werden (siehe Tabelle 13.5).

### 9.4.2 Evaluierung des Designs

Im folgenden werden mögliche Änderungen von Designentscheidungen bei der Realisierung von Kapazitätsconstraints und Serialisierern in Oz und ihre Auswirkungen untersucht.

In Tabelle 9.3 sind für die aufgeführten Probleme Resultate für den Optimalitätsbeweis mit der Beweisphase angegeben. Die Spalte *Fails* gibt die Anzahl der Fehlerknoten im entstehenden Suchbaum und die Spalte *CPU* die Laufzeit in Sekunden an. Für die Spalte *Oz* wurde die Kombination • verwendet.

**Propagierer anstatt Basisconstraints?** Da ein Propagierer für einen Kapazitätsconstraint nicht bei jeder Ausführung die gleichen Aufgabenintervalle konstruieren muß, könnte es vorteilhaft sein, Propagierer anstatt Basisconstraints in geeigneten Fällen zum Berechnungsraum hinzuzufügen. Für Spalte $Oz^{\dagger}$ wurde der Propagierer für den Kapazitätsconstraint mit dem Zwei-Phasen-Algorithmus so abgeändert, daß Propagierer (anstatt Basisconstraints) zum Berechnungsraum hinzugefügt werden, wenn eine Aufgabe vor allen anderen oder nach allen anderen plaziert werden kann (siehe Proposition 12 und 13 auf Seite 65). In der Tabelle erkennt man, daß diese Änderung an der Zahl der nötigen Fehlerknoten praktisch nichts ändert, aber im Schnitt etwa 15% mehr Rechenzeit kostet. Insgesamt werden durch diese Änderung nämlich etwa zehnmal

mehr Propagierer erzeugt, von denen die meisten jedoch rasch aufhören zu existieren, da der realisierte Constraint vom Constraintspeicher subsumiert ist.

**Redundante Propagierer und Distribuierung?** In Spalte $Oz^{\ddagger}$ ist hingegen der aufgabenorientierte Serialisierer geändert worden. In Abschnitt 7.3.4 haben wir gezeigt, daß der Serialisierer aus Abschnitt 6.3 in Oz anders implementiert ist als in Abschnitt 6.3 vorgestellt. Es werden nämlich in einem Distribuierungsschritt sowohl redundante Propagierer zum Berechnungsraum hinzugefügt als auch Ordnungsentscheidungen durch Distribuierung getroffen. Eine Alternative ist es, erst dann zu distribuieren, wenn die Strategie selbst keine Propagierer mehr zum Berechnungsraum hinzufügen kann; also genau so zu verfahren wie es in Abschnitt 6.3 geschildert wird (in Oz wird für das Hinzufügen der redundanten Propagierer aber kein zusätzlicher Wahlpunkt verwendet, da man die Propagierer direkt ohne Duplizieren zu dem aktuellen Berechnungsraum hinzufügen kann). Auf diese Weise könnte mehr Information zu einer fundierteren Distribuierung zur Verfügung stehen. In der Tabelle erkennt man aber, daß sich die Anzahl der Fehlerknoten nur in sechs Fällen wenig verbessert und in vier Fällen sogar verschlechtert. Die Laufzeit steigt im Schnitt sogar um etwa 30% an, da nun insgesamt mehr Aufgabenintervalle berechnet werden.

**Tabelle 9.3** Auswirkungen von geänderten Kapazitätsconstraints und Serialisierern

| | Oz | | $Oz^{\dagger}$ | | $Oz^{\ddagger}$ | | $Oz^{\dagger}$/Oz | $Oz^{\ddagger}$/Oz |
|---|---|---|---|---|---|---|---|---|
| Problem | Fails | CPU | Fails | CPU | Fails | CPU | CPU | CPU |
| MT10 | 1 799 | 28.7 | 1 799 | 33.3 | 1 830 | 37.3 | 1.16 | 1.30 |
| ABZ5 | 1 431 | 23.8 | 1 431 | 26.7 | 1 420 | 31.5 | 1.12 | 1.32 |
| ABZ6 | 148 | 2.3 | 148 | 2.7 | 149 | 3.3 | 1.17 | 1.43 |
| La19 | 1 066 | 17.7 | 1 066 | 20.1 | 1 057 | 24.7 | 1.14 | 1.40 |
| La20 | 881 | 13.9 | 880 | 15.7 | 879 | 19.0 | 1.13 | 1.37 |
| ORB1 | 7 528 | 120.9 | 7 527 | 139.4 | 7 483 | 149.1 | 1.15 | 1.23 |
| ORB2 | 425 | 7.2 | 425 | 8.0 | 423 | 9.5 | 1.11 | 1.32 |
| ORB3 | 22 579 | 339.4 | 22 577 | 387.4 | 22 657 | 420.7 | 1.14 | 1.24 |
| ORB4 | 1 034 | 15.7 | 1 034 | 17.7 | 1 040 | 20.2 | 1.13 | 1.29 |
| ORB5 | 869 | 14.4 | 869 | 16.5 | 863 | 19.1 | 1.15 | 1.33 |

**Serialisierer ohne redundante Propagierer?** Tabelle 9.4 enthält die Ergebnisse für die Optimalitätsbeweise, wenn der aufgabenorientierte Serialisierer aus Abschnitt 6.3 erneut leicht abgeändert wird (die entsprechenden Laufzeiten verändern sich proportional zu der Fehlerknotenzahl und werden deshalb nicht aufgeführt). Zum Vergleich enthält die Spalte *Oz* noch einmal die Ergebnisse der Kombination •. Für die Spalte $Oz(EF_1)$ trägt der Serialisierer selbst überhaupt keine Propagierung durch das Hinzufügen redundanter Propagierer bei. Für die meisten Probleme steigt die Zahl der notwendigen Fehlerknoten (teilweise drastisch) an. Dies belegt die Wichtigkeit der zusätzlichen Propagierung durch den Serialisierer. Beachte, daß der Serialisierer Aufgabenintervalle verwendet, die mehr Edge-Finding Anwendungen ermöglichen, als der Zwei-Phasen-Algorithmus (siehe auch die Diskussion in Abschnitt 6.3.1). Für Spalte $Oz(EF_2)$ fügt der Serialisierer keine redundanten Propagierer sondern nur redundante Basisconstraints zum Berechnungsraum hinzu, die zum Aufrufzeitpunkt des Serialisierers ableitbar sind. Nun verringert sich teilweise sogar die Zahl der nötigen Fehlerknoten im Vergleich zur Spalte *Oz*, steigt jedoch

für das Problem ORB3 unbefriedigend an. Obwohl der Serialisierer also selten ausgeführt wird, können auch die Basisconstraints den Suchraum stark einschränken.

In Spalte *Oz(Disj)* wird die Kombination $\diamond$ verwendet. Findet durch den Serialisierer wiederum gar keine Propagierung statt (Spalte *Oz(Disj$_1$)*), so sind die Ergebnisse sehr schlecht. Die Optimalität kann teilweise sogar mit einer Million Fehlerknoten nicht bewiesen werden. Werden nun Basisconstraints durch den Serialisierer zum Berechnungsraum hinzugefügt (Spalte *Oz(Disj$_2$)*), so ist das Ergebnis ähnlich zu *Oz(Disj)*, aber wieder wird ORB3 sehr viel schlechter gelöst. Gerade diese Meßreihe macht besonders deutlich, wie wichtig es ist, daß der Serialisierer redundante Propagierer mit Edge-Finding (oder zumindest Basisconstraints) zum Berechnungsraum hinzufügt.

**Tabelle 9.4** Veränderter aufgabenorientierter Serialisierer

| | Oz | Oz(EF$_1$) | Oz(EF$_2$) | Oz(Disj) | Oz(Disj$_1$) | Oz(Disj$_2$) |
|---|---|---|---|---|---|---|
| Problem | Fails | Fails | Fails | Fails | Fails | Fails |
| MT10 | 1 799 | 2 427 | 1 778 | 3 345 | 477 725 | 3 357 |
| ABZ5 | 1 431 | 1 714 | 1 428 | 2 288 | 8 067 | 2 285 |
| ABZ6 | 148 | 189 | 142 | 239 | 1 860 | 229 |
| La19 | 1 066 | 1 015 | 1 016 | 1 696 | 3 988 | 1 579 |
| La20 | 881 | 871 | 872 | 3 127 | 20 977 | 3 157 |
| ORB1 | 7 528 | 16 928 | 7 594 | 14 123 | $>$ 1 000 000 | 13 734 |
| ORB2 | 425 | 455 | 435 | 680 | 3 263 | 687 |
| ORB3 | 22 579 | 35 038 | 29 780 | 35 744 | $>$ 1 000 000 | 45 833 |
| ORB4 | 1 034 | 1 213 | 1 009 | 1 908 | 20 488 | 1 984 |
| ORB5 | 869 | 949 | 869 | 1 566 | 30 095 | 1 550 |

### 9.4.3 Laufzeitanalyse

Eine Analyse der Laufzeiten für Optimalitätsbeweise zeigt, daß der Anteil der Kapazitätsconstraints an der Gesamtlaufzeit für Kombination $\bullet$ etwa 43% beträgt. Der Anteil der Distribuierung an der Gesamtlaufzeit beträgt etwa 23%. Die Präzedenzconstraints machen rund 9% aus. Wird wie in $\otimes$ ein ressourcenorientierter Serialisierer verwendet, bleibt der Anteil der Kapazitätsconstraints bei rund 43%. Der Anteil des Serialisierers sinkt auf 3%. Dafür steigt der Anteil für die Präzedenzconstraints auf rund 13%.

Für die Optimierungsphase und die Kombination $\bullet$ sinkt der Anteil der Kapazitätsconstraints an der Laufzeit auf etwa 31% ab. Der Anteil der Distribuierung bzw. der Präzedenzconstraints liegt bei 24% bzw. 7%. Der Anteil an sonstigem (funktionalem und objektorientiertem) Oz-Code beläuft sich auf rund 18%. Bei einer Kombination wie $\otimes$ liegt der Anteil der Kapazitätsconstraints bei etwa 32%, der Anteil der Serialisierer bei etwa 2% und der Anteil der Präzedenzconstraints bei rund 12%. Der Anteil an sonstigem Oz-Code beträgt rund 20%. Der Anteil des Oz-Codes an der Laufzeit steigt jedoch stark an, wenn praktisch nur der gierige Algorithmus (wie zum Beispiel bei Problem LA20) zum Zuge kommt.

Der Speicherverbrauch in Oz ist für diese Probleme relativ gering. Sowohl in der Optimierungsphase als auch in der Beweisphase wird zur Lösung der Standardprobleme etwa ein MB akti-

ver Daten benötigt. Im Schnitt wird in der Beweisphase für die Kombination $\bullet$ etwa 15% der Gesamtlaufzeit für Kopieren von Berechnungsräumen und etwa 5% der Zeit für Speicherbereinigung verwendet. Für Kombination $\otimes$ steigen diese Zahlen auf 25% für Kopieren und 9% für Speicherbereinigung an. Etwa die gleichen Prozentzahlen gelten auch für die Optimierungsphase. Weshalb für aufgabenorientierte Serialisierer weniger kopiert wird, kann daran liegen, daß hier mehr Propagierer subsumiert sind als im ressourcenorientierten Fall. Zudem werden im ressourcenorientierten Fall größere Datenstrukturen in Oz verwaltet (die Liste der Kandidaten, die zuerst bzw. zuletzt plaziert werden können).

## 9.5 Verwandte Arbeiten

Für einen Vergleich beschränken wir uns auf constraintbasierte Arbeiten. Das erste constraintbasierte System zum Lösen von Job-Shop-Schedulingproblemen war ISIS [FS84, Fox94]. ISIS konnte auch mit weichen Constraints umgehen (siehe Abschnitt 8.4) und basierte auf einer joborientierten Vorgehensweise. Daraus ging OPIS [SOP+86, Smi94] hervor, in dem Ressourcen als Engpässe des Schedulings betrachtet werden und ressourcenorientiert vorgegangen wird. Eine Weiterentwicklung ist MICRO-BOSS [Sad91, Sad94], worin weder ausschließlich alle Aufgaben eines Jobs noch einer Ressource vollständig plaziert werden. Stattdessen wird dynamisch die Ressourcenauslastung untersucht, daraufhin eine Aufgabe ausgewählt und nach einer speziellen Heuristik plaziert. Alle diese Systeme arbeiten nicht mit Serialisierern, sondern bestimmen für die Aufgaben Startzeitpunkte. Die in [Sad91] angegebenen Probleme sind sehr leicht mit den im Oz-Scheduler verfügbaren Techniken lösbar. Umgekehrt, sind die MICRO-BOSS-Techniken nicht in der Lage, die schweren Job-Shop-Probleme wie MT10 aus den vorangehenden Abschnitten zu lösen.

Der Ansatz aus [Sad91] wird in [BDSF97b, BDSF97a] mit Serialisierungstechniken und Edge-Finding verbunden. Insbesondere wird noch *limited discrepancy search (LDS)* [HG95] zum schnellen Finden guter Lösungen verwendet. In [BDSF97b, BDSF97a] wird versucht, von der optimalen Schedulelänge ausgehend eine Lösung zu finden. Gelingt dies innerhalb einer vorgegebenen Zeit nicht, wird versucht für eine etwas größere Schedulelänge eine Lösung zu finden usw. In der Praxis kommt es natürlich selten vor, daß man die optimale Schedulelänge bereits kennt. Die erzielten Ergebnisse können jedoch alle mit den hier vorgestellten Techniken (teilweise besser) erzielt werden.

In [Col96] wird ein Branch-and-Bound-Ansatz zur Lösung von Job-Shop-Problemen vorgestellt, der eine Art von Edge-Finding und einen speziellen ressourcenorientierten Serialisierer benutzt. Mit diesem Ansatz können die Standardprobleme aus Tabelle 13.5 zwar gut, größere Probleme aber nicht gelöst werden [Col96].

CLAIRE [CL94b] ist eine funktionale und objektorientierte Programmiersprache, die auch regelbasiertes Schließen ermöglicht. Die Sprache wurde speziell zur Lösung von komplexen Optimierungsproblemen entworfen. CLAIRE-Programme lassen sich nach C++ übersetzen, so daß eine effiziente Ausführung gewährleistet ist. Constraintpropagierung wird durch das Konzept von Produktionsregeln unterstützt. Sogenannte *worlds* ermöglichen die Programmierung von Suchstrategien (ähnlich wie Berechnungsräume in Oz). Eine Erweiterung der disjunktiven Schedulingtechniken in CLAIRE ist die Bibliothek CLAIRE SCHEDULE [PB97], wodurch kumulatives

Scheduling von unterbrechbaren Aufgaben ermöglicht wird (siehe auch [PB96]). Eine grafisch unterstützte Werkbank vergleichbar dem Oz-Scheduler existiert jedoch nicht.

ILOG SCHEDULER [ILOG96a] ist eine C++-Klassenbibliothek zur Lösung von Schedulingproblemen. Diese Bibliothek basiert auf dem System ILOG SOLVER [ILOG96b] zur Lösung von kombinatorischen Problemen (siehe Kapitel 13 für mehr Informationen). Einen Vergleich einiger Schedulingtechniken in ILOG SCHEDULER enthält [Bap94]. ILOG SCHEDULER ist ein kommerzielles Produkt und bietet Funktionalität zur Lösung von disjunktiven und kumulativen Schedulingproblemen an. Ein grafik-basiertes Werkzeug wie der Oz-Scheduler existiert nicht. Jedoch hat ein Benutzer Visualisierungswerkzeuge von ILOG zur Auswahl, um solche Werkzeuge prinzipiell bauen zu können. Auch heuristische Verfahren mit Iterativer Verbesserung, wie sie im Oz-Scheduler verwendet werden, gibt es in ILOG SCHEDULER nicht.

Ein Vergleich der im Oz-Scheduler verfügbaren Schedulingtechniken ist so umfassend bisher noch nicht durchgeführt worden. In [BDSF97b, BDSF97a] werden zwar verschiedene Heuristiken zum Finden guter Lösungen verglichen. Es wird jedoch nicht der Optimalitätsbeweis untersucht (siehe auch die obigen Bemerkungen zu [BDSF97b, BDSF97a]) und es werden keine Laufzeitvergleiche gemacht. In [Bap94] wird Edge-Finding nach [Nui94], Edge-Finding nach [CP89] und die Technik aus [ELT91] verglichen und sowohl die Anzahl der Fehlerknoten als auch Laufzeiten angegeben.

# Kapitel 10

# Taktgesteuerte Echtzeitsysteme

Diese Fallstudie entstand aus einem Kooperationsprojekt mit Daimler-Benz [SW97, SW98] und erschließt der Constraintprogrammierung ein neues Anwendungsgebiet. Zur Lösung des zugrundeliegenden Problems ist zum einen Expressivität für Propagierer und Distribuierer gefordert. Zum anderen ist wegen der Größe des enthaltenen Schedulingproblems besonders Effizienz gefragt. So ist zum Beispiel ein Schedulingproblem zu lösen, in dem auf einer Ressource mehr als 1 800 Aufgaben zu plazieren sind, deren Startzeiten zwischen Null und 6 Millionen liegen können. Somit zeigt diese Fallstudie eindrucksvoll, daß selbst sehr große Probleme mit Oz gelöst werden können.

In Abschnitt 10.2 wird das zugrundeliegende Problem beschrieben. Im folgenden Abschnitt 10.3 wird das Problem modelliert, und es wird auf die verwendeten Projektoren und Distribuierungsstrategien eingegangen. Abschnitt 10.4 faßt die empiririschen Ergebnisse bei der Lösung der vorliegenden Probleme zusammen. Die folgenden Abschnitte enthalten eine Beschreibung der theoretischen Komplexität des Problems, die Funktionalität des implementierten Prototyps und einen kurzen Ausblick.

## 10.1 Einleitung

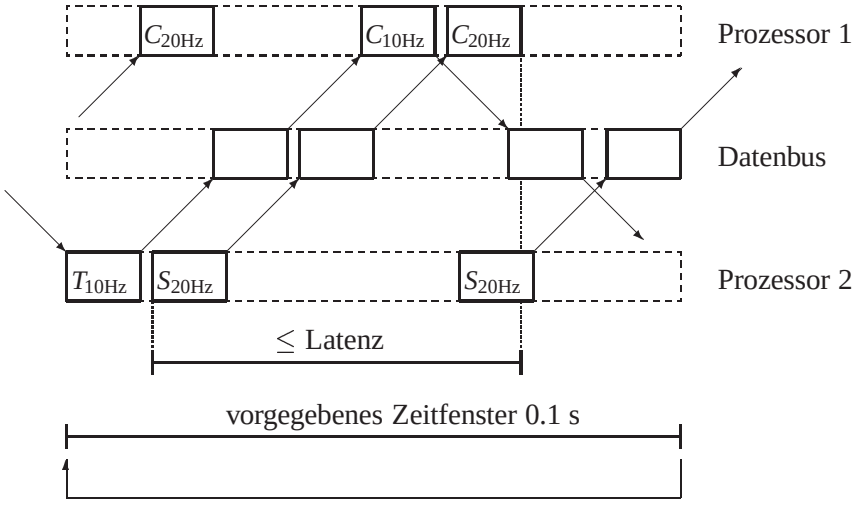
In jüngster Zeit gibt es eine wachsende Zahl von verteilten Echtzeitanwendungen, bei denen die Berechnung von Prozessen und die dazugehörige Kommunikation einem regulären Muster folgen muß. Genau für diese Art von Anwendungen wurde eine spezielle Klasse von sogenannten *taktgesteuerten* (engl. *time-triggered*) Architekturen entwickelt [Kop91]. Das gesamte Systemverhalten dieser Architekturen wird durch ein periodisches Taktsignal kontrolliert. Taktgesteuerte Architekturen sind besonders gut für sicherheitskritische Anwendungen geeignet. Erfolgreiche Anwendungsgebiete dieses speziellen Architekturtyps finden sich zum Beispiel in der Autoelektronik oder der Luft- und Raumfahrtindustrie.

Das Hauptproblem bei einer taktgesteuerten Architektur ist es, vor der tatsächlichen Ausführung einen passenden Schedule für Prozesse und die Kommunikation zu generieren. Dieses *off-line* Schedulingproblem ist insofern untypisch, als daß eine ständig wiederholte Berechnung und Kommunikation auf ein einzelnes Zeitfenster abgebildet werden müssen. Genau dieses Zeitfen-

ster bestimmt durch seine Wiederholung das Gesamtverhalten des Systems.

Als Beispiel nehmen wir einen Thermostatprozeß $T_{10Hz}$ an, der mit einer Frequenz von genau 10Hz die gemessene Temperatur an einen Kontrollprozeß $C_{10Hz}$ schickt. Umgekehrt schickt $C_{10Hz}$ mit genau der gleichen Frequenz Information über eine notwendige Temperaturänderung zurück. Eine solche zyklische Abhängigkeit zwischen Prozessen ist explizit erlaubt. Wenn zwei Anwendungsprozesse auf verschiedenen Prozessoren laufen, können sie nur über einen gemeinsamen Datenbus kommunizieren. Die Architektur enthält genau einen solchen Datenbus, über den nur eine Nachricht gleichzeitig übertragen werden kann. In Abbildung 10.1 wird angenommen, daß die Kommunikation zwischen Thermostat und Kontrollprozeß eine solche Kommunikation zwischen verschiedenen Prozessoren ist. Weiter nehmen wir an, daß es einen zweiten Kontrollprozeß $C_{20Hz}$ gibt, der mit einer Frequenz von 20Hz Daten von einem Sensor $S_{20Hz}$ erhält. In diesem Fall soll die Kommunikation nur in eine Richtung gehen. Wir nehmen an, daß $S_{20Hz}$ und $C_{20Hz}$ auf verschiedenen Prozessoren laufen, während $S_{20Hz}$ und $T_{10Hz}$sowie $C_{20Hz}$ und $C_{10Hz}$ auf dem gleichen Prozessor laufen. Auf jedem Prozessor darf zu jedem Zeitpunkt höchstens ein Anwendungsprozeß laufen. Dabei darf kein Prozeß verdrängt werden. Das Schedulingproblem besteht dann darin, diese unterschiedlichen Kommunikationsmuster auf ein einzelnes Zeitfenster abzubilden. Wird dieses Zeitfenster ständig wiederholt, sollte das gewünschte Gesamtverhalten des Systems resultieren. Abbildung 10.1 zeigt ein gültiges Zeitfenster, das für das Beispiel das gewünschte Verhalten erzielt.

**Abbildung 10.1** Zeitfenster




Weiterhin können auch zeitliche Constraints zwischen Prozessen auf verschiedenen Prozessoren vorkommen, sogenannte *Latenzconstraints* (engl. *latency constraints*). Abbildung 10.1 enthält einen solchen Constraint zwischen $S_{20Hz}$ und $C_{20Hz}$. Dieser Constraint definiert eine zulässige obere Schranke für die Zeit, die zwischen dem Start von $S_{20Hz}$ und dem Ende von $C_{20Hz}$ vergehen darf, so daß die zugehörige Nachricht zwischen diesen Grenzen gesendet und empfangen werden kann. Dieser Constraint garantiert, daß wichtige Informationsverarbeitung in einem gegebenen Zeitrahmen abgeschlossen wird.

Diese ungewöhnlichen Eigenschaften machen solche Schedulingprobleme für taktgesteuerte Ar-

chitekturen zu einem herausfordernden neuen Anwendungsgebiet für Constraintprogrammierung. Es existieren überhaupt nur wenige Ansätze, solche Probleme anzugehen. Einen Überblick zu Anwendungen speziell in der Automobilindustrie bietet [Eri97].[1] Die mit Constrainttechniken angegangenen Echtzeitanwendungen sind eher kleine Schedulingprobleme, bei denen jedoch der Schedule selbst zur Echtzeit berechnet werden muß [FS95] (siehe [Wal96] für einen Überblick).

Es ist das erste Mal, daß Constrainttechniken zur Bearbeitung von taktgesteuerten Architekturen verwendet werden. Das betrachtete Problem stammt aus einer umfangreichen Anwendung im Daimler-Benz-Konzern. Es enthält endliche Bereiche von 6 Millionen verschiedenen Startzeiten und mehr als 2 000 Prozesse und Nachrichten. Während der Lösungssuche werden mehr als eine Million Propagierer erzeugt.

In dieser Fallstudie werden verschiedene Techniken zur Lösung traditioneller Schedulingprobleme und von Echtzeitanwendungen auf ihre Eignung für taktgesteuerte Architekturen verglichen. Es wird gezeigt, welche Techniken erfolgreich angewendet werden können und welche nicht. Globale Constraints werden eingesetzt, um Laufzeit und vor allem Speicherbedarf zu reduzieren. Suchstrategien aus dem Operations Research und von Echtzeitanwendungen werden getestet. Nur eine relativ neue Strategie aus dem Operations Research ist bei der Problemlösung erfolgreich. Es reicht aus, eine totale Ordnung der Prozesse und Nachrichten zu finden, um einen gültigen Schedule zu berechnen, also jeweils eine Serialisierung zu finden. Das übliche Kriterium der Auslastung ist für eine Engpaß-Analyse (engl. bottleneck-analysis) unzureichend. Um Probleme einer solchen Größenordnung zu lösen, müssen redundant gewordene Constraints so früh wie möglich von der Speicherbereinigung beseitigt werden. Das präsentierte Vorgehen erlaubt es, alle uns vorliegenden Probleme zu lösen.

## 10.2 Das Schedulingproblem

In diesem Abschnitt wird eine leicht vereinfachte Version des Schedulingproblems geschildert.[2] Insbesondere betrachten wir nur den Fall synchroner Kommunikation, bei der Sender und Empfänger von Nachrichten die gleiche Frequenz besitzen. Asynchrone Kommunikation, bei der der Sender eine vom Empfänger verschiedene Frequenz besitzt, wird ausführlich in [SW97] behandelt.

Ein *Mehrprozessorsystem* besteht aus einer endlichen Zahl von verschiedenen *Prozessoren*. Ein *Prozeß* ist höchstens einem Prozessor zugeordnet, seinem sogenannten *Wirt* (engl. *host*). Ein Prozeß kann nur von seinem Wirt ausgeführt werden.

Jeder Prozeß $P$ hat eine nichtnegative *Ausführungszeit* $dur(P)$. Eine einzelne Ausführung von $P$ wird mit $P_i$, $i \geq 1$, bezeichnet. Jede solche Prozeßausführung $P_i$ hat eine nichtnegative *Startzeit* $start(P_i)$. Das *Ende* $compl(P_i)$ von $P_i$ ist dann wie folgt eindeutig bestimmt.

$$compl(P_i) \quad = \quad start(P_i) + dur(P). \tag{10.1}$$

[1] Eine weitergehende Literaturanalyse muß leider unterbleiben, da der Auftraggeber des Projekts strikte Vertraulichkeit über das Anwendungsgebiet und damit seine Identität verlangt hat. Deshalb soll hier nicht durch spezifische Literaturverweise auf den Auftraggeber geschlossen werden können.

[2] Das reale Problem enthält zum Beispiel weitere Synchronisationsnachrichten. Weiterhin variiert die Länge von Nachrichten mit ihrer Funktion, es muß ein gewisser Abstand zwischen Nachrichten zugesichert werden und einiges mehr.

Dabei nehmen wir implizit an, daß eine Prozeßausführung nicht unterbrochen werden darf.

Ein Prozeß $P$ kann *periodisch* oder *aperiodisch* sein. Ist er periodisch, wird er mit einer vorgegebenen Frequenz ausgeführt. Der reziproke Wert dieser Frequenz wird die *Periode* $period(P)$ von $P$ genannt. Eine Frequenz von zum Beispiel 10 Hertz ergibt somit eine Periode von 0.1 Sekunden. Der zeitliche Abstand zwischen den Startzeitpunkten zweier aufeinanderfolgender Instanzen eines bestimmten Prozesses muß immer mit der Periode des Prozesses übereinstimmen. Somit ergibt sich die folgende Periodizitätsbedingung.

$$start(P_i) \quad = \quad start(P_{i-1}) + period(P), \text{ für } i \geq 2. \tag{10.2}$$

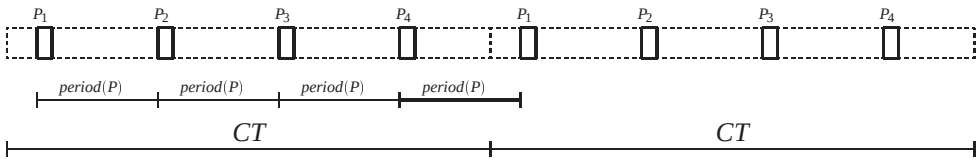
Diese Bedingung macht natürlich nur Sinn, wenn

$$period(P) \quad \geq \quad dur(P) \tag{10.3}$$

gilt. In der Praxis ist die Periode eines Prozesses oft sehr viel größer als seine Ausführungszeit.

Wir nehmen an, daß die Gesamtkontrolle des Multiprozessorsystems vor dem tatsächlichen Ablauf *a priori* in einem festen Zeitfenster, dem *Wiederholungsfenster*, organisiert ist, das ständig wiederholt wird. Die Länge dieses Fensters heißt die *Zykluszeit* $CT$. Diese Länge sollte so gewählt werden, daß ein periodischer Prozeß in ein Wiederholungsfenster paßt ohne aus der Phase zu laufen. Es muß aber nicht nur die Periodizitätsbedingung (10.2) innerhalb eines Wiederholungsfensters eingehalten werden, sondern eine ähnliche Bedingung muß auch für die verketteten Wiederholungsfenster gelten (siehe Abbildung 10.2).

**Abbildung 10.2** Verkettung zweier Wiederholungsfenster




Dies kann dadurch erreicht werden, daß für die Zykluszeit das kleinste gemeinsame Vielfache der vorkommenden Perioden gewählt wird (ein häufig gewähltes Verfahren für das betrachete Anwendungsgebiet; siehe zum Beispiel [CA95]).

Sei $P$ ein beliebiger Prozeß. Dieser Prozeß muß $n = \frac{CT}{period(P)}$-mal innerhalb eines Wiederholungsfensters ausgeführt werden. Diese Zahl wird als $\#P$ bezeichnet. Wenn alle Ausführungen von $P$ die Periodizitätsbedingung (10.2) erfüllen, beträgt der zeitliche Abstand zwischen $start(P_1)$ und $start(P_n)$ genau $(n-1) \cdot period(P)$. Aber dann beträgt der zeitliche Abstand zwischen dem Startzeitpunkt der letzten Ausführung von $P$ im Wiederholungsfenster und dem Startzeitpunkt seiner ersten Ausführung im folgenden Fenster $CT - (n-1) \cdot period(P) = period(P)$. Die Periodizitätsbedingung für die Verkettung zweier Wiederholungsfenster ist somit eine Konsequenz von (10.2) und $\#P = \frac{CT}{period(P)}$.

Ein aperiodischer Prozeß $P$ wird wie ein Prozeß mit der Periode $CT$ behandelt. Wir nehmen an, daß alle Ausführungen eines Prozesses innerhalb eines Wiederholungsfensters liegen. Deshalb muß folgende obere Schranke für das Ende eines Prozesses gelten.

$$compl(P_i) \quad \leq \quad CT. \tag{10.4}$$

Wenn zwei Prozesse auf dem gleichen Wirt ausgeführt werden, darf sich ihre Ausführung nicht zeitlich überlappen. Die Ausführungen müssen also wie folgt serialisiert werden.

$$\begin{array}{c}\text{entweder } compl(P_i) \leq start(Q_j) \text{ oder } compl(Q_j) \leq start(P_i) \\ \text{wenn } P \text{ und } Q\ (P \neq Q) \text{ den gleichen Wirt haben und } i \neq j \text{ gilt.}\end{array} \tag{10.5}$$

Beachte, daß für einen einzelnen Prozeß die Serialisierung schon durch die Periodizitätsbedingung (10.2) garantiert ist. Der obige disjunktive Constraint beinhaltet deshalb nicht den Fall, daß $P$ und $Q$ identisch sind. Ein solcher Constraint ist nichts anderes als ein Kapazitätsconstraint für disjunktive Schedulingprobleme (siehe Abschnitt 5.1).

### 10.2.1 Kommunikation zwischen Prozessoren

Haben zwei Prozesse unterschiedliche Wirte, müssen sie durch eine *Nachricht* über einen gemeinsamen Datenbus kommunizieren. Es ist nur ein Datenbus verfügbar, mit dem alle Prozessoren verbunden sind. Zwei Prozesse mit dem gleichen Wirt können hingegen direkt miteinander kommunizieren.

Eine Nachricht hat genau einen *Sender*, kann aber mehrere *Empfänger* haben.[3] Sender und Empfänger sind jeweils bestimmte Prozesse. Hat mindestens ein Empfänger einen anderen Wirt als der Sender, muß die Nachricht über den Datenbus übertragen werden. Solche Nachrichen heißen *Inter-Prozessor-Nachrichten* (im Gegenatz zu den *Intra-Prozessor-Nachrichten*).

Mit jeder Nachricht $M$ ist eine nichtnegative Zahl $dur(M)$ assoziiert; die *Übertragungszeit* von $M$. Ist $M$ eine Intra-Prozessor-Nachricht, hat $dur(M)$ den Wert Null.

Es gibt weiterhin eine besondere Art von Inter-Prozessor-Nachricht, die zur Synchronisation der lokalen Uhren der Prozessoren dient und *Resynchronisationsnachricht* genannt wird. Wir nehmen jeweils einen sonst nicht vorkommenden Prozessor als Sender und Empfänger an. Die Übertragungszeit ist sehr kurz und die Frequenz des Senders sehr hoch (einige 1 000 Hz).

Eine einzelne Übertragung einer Nachricht $M$ wird als $M_i$ bezeichnet, $i \geq 1$. Jede solche Übertragung hat einen Startzeitpunkt $start(M_i)$ und eine Abschlußzeit $compl(M_i)$.

$$compl(M_i) \quad = \quad start(M_i) + dur(M). \tag{10.6}$$

Ist $S$ der Sender von $M$, muß es eine Folge von Übertragungen $M_1, \ldots, M_n$ mit $n = \#S$ geben. Dies gilt natürlich nur für den synchronen Fall, wenn alle Empfänger die gleiche Frequenz wie der Sender haben. Im synchronen Fall werden Nachrichten mit der gleichen Frequenz wie der Sender gesendet und dürfen nicht die Zykluszeit überschreiten:

$$start(M_i) \quad = \quad start(M_{i-1}) + period(S), \text{ für } i \geq 2, \tag{10.7}$$

$$compl(M_i) \quad \leq \quad CT. \tag{10.8}$$

Es ist nicht notwendig, daß eine Nachricht unmittelbar nach dem Ende ihres Senders übertragen wird. Eine Nachricht kann solange gepuffert werden wie keine neue Instanz dieser Nachricht

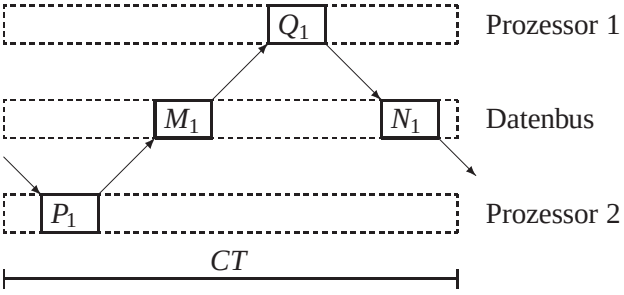
[3] In der Praxis ist es möglich, mehrere Sender für eine Nachricht zur Auswahl zu spezifizieren. Dies ist nützlich, falls zur Laufzeit des Systems ein Prozeß aus technischen Gründen nicht senden kann.

erzeugt wird. Eine Nachrichtenübertragung darf aber nicht vor dem Ende ihres Senders begonnen werden:

$$compl(S_i) \quad \leq \quad start(M_i). \tag{10.9}$$

Es gibt jedoch keinen vergleichbaren Präzedenzconstraint der Form $compl(M_i) \leq start(R_j)$ zwischen einer Nachricht $M_i$ und einem ihrer Empfänger $R_j$. Denn der Empfänger einer Nachricht kann bis zu dem Anfang des folgenden Wiederholungsfensters verzögert werden. Für ein zyklisches Kommunikationsmuster ist dies sogar unumgänglich. Nehmen wir das spezielle zyklische Kommunikationsmuster in Abbildung 10.3 als Beispiel.

**Abbildung 10.3** Verzögern des Empfängers einer Nachricht




Hier sendet $P$ eine Nachricht an $Q$, der wiederum eine zweite Nachricht an $P$ zurückschickt. Haben die beiden Prozesse die gleiche Frequenz $\frac{1}{CT}$, kann der Empfänger der zweiten Nachricht nicht nach der Übertragung dieser Nachricht im selben Wiederholungsfenster plaziert werden (ansonsten müßte der Empfänger die doppelte Frequenz wie der Sender haben). Somit kann die zweite Nachricht nur im folgenden Wiederholungsfenster empfangen werden.

Andererseits ist es implizit erforderlich, daß die Übertragung einer Nachricht vor dem Start der nachfolgenden Ausführung ihres Senders abgeschlossen ist:

$$compl(M_i) \quad \leq \quad start(S_{i+1}).$$

Dies ist eine Konsequenz aus (10.7)–(10.9) und der Tatsache, daß es genausoviele Übertragungen von $M$ wie Ausführungen von $S$ gibt. Ansonsten würde der Startzeitpunkt der letzten Übertragung von $M$ die obere Schranke $CT$ übersteigen und damit (10.8) verletzen.

Wie die Ausführungen der Prozesse auf dem gleichen Wirt, müssen auch alle Übertragungen von Inter-Prozessor-Nachrichten serialisiert werden:

$$\begin{gathered}\text{entweder } compl(M_i) \leq start(N_j) \text{ oder } compl(N_j) \leq start(M_i) \\ \text{wenn } M \text{ und } N\ (M \neq N) \text{ beides Inter-Prozessor-Nachrichten sind und } i \neq j \text{ gilt.}\end{gathered} \tag{10.10}$$

Für Intra-Prozessor-Nachrichten ist ein solcher Kapazitätsonstraint nicht notwendig, da ihre Ausführungszeit Null beträgt.

### 10.2.2 Latenzconstraints

Für ein Echtzeitsystem ist es wichtig, daß kritische Information garantiert innerhalb gewisser Zeitgrenzen verarbeitet wird. Sendet ein Prozeß eine Nachricht zu einem zweiten Prozeß, so sollte die gesamte Informationsverarbeitung ein gewisses Zeitlimit nicht überschreiten. Für die hier betrachtete Anwendung bedeutet dies, daß es eine obere Schranke für die Zeit zwischen der Erzeugung einer Information durch den Sender und ihrer vollständigen Verarbeitung durch den Empfänger gibt, wobei der Informationsaustausch zwischen diesen Zeitpunkten erfolgen muß. Eine solche obere Schranke wird *Latenzzeit* genannt. Der spezielle Wert der Latenzzeit hängt nicht nur von der Nachricht ab, sondern auch von dem speziellen Empfänger. Dieser Wert wird im folgenden als $latency(M,R)$ bezeichnet, wobei $R$ einer der Empfänger der Nachricht $M$ ist. Ist $S$ der Sender dieser Nachricht, ergibt sich der folgende Constraint.

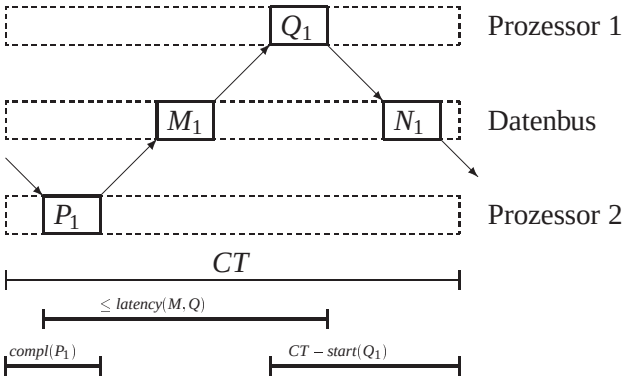
$$compl(R_i) - start(S_i) \quad \leq \quad latency(M,R).$$

Abbildung 10.4 zeigt ein Beispiel für einen Latenzconstraint für den Abstand zwischen $start(P_1)$ und $compl(Q_1)$.

Dieser Typ von Latenzconstraint setzt voraus, daß die Übertragung $M_i$ von der Ausführung $R_i$ empfangen wird. Dies kann aber nur für synchrone Kommunikation garantiert werden. Die Erweiterung für asynchrone Kommunikation wird in [SW97] behandelt.

Diese Art des Latenzconstraints deckt jedoch auch im Falle synchroner Kommunikation nicht alle relevanten Situationen ab. In Abbildung 10.4 sendet $P$ nicht nur die Nachricht $M$ an $Q$, sondern $Q$ sendet auch $N$ and $P$ zurück. Der Empfänger der Übertragung von $N$ kann aber nicht nach Abschluß dieser Übertragung ausgeführt werden. Stattdessen muß die Ausführung des Empfängers auf den Beginn des nachfolgenden Wiederholungsfensters verzögert werden.

**Abbildung 10.4** Latenzconstraints




In diesem Fall ist eine ausreichende Bedingung, daß $CT - start(Q_1) + compl(P_1)$ nicht größer als $latency(N,Q)$ sein darf (siehe Abbildung 10.4). Für den Latenzconstraint muß somit die Verkettung von Wiederholungsfenstern berücksichtigt werden. Sei dazu $S_i$ der Sender einer Nachricht $M_i$. Für jeden Empfänger $R_i$ dieser Nachricht muß gelten:

$$\begin{array}{ll} \text{Wenn} & start(R_i) \geq compl(M_i) \\ \text{dann} & compl(R_i) - start(S_i) \leq latency(M,R) \\ \text{sonst} & CT - start(S_i) + compl(R_1) \leq latency(M,R). \end{array} \tag{10.11}$$

Solch ein Latenzconstraint wird natürlich nur dann verwendet, wenn die übertragene Information auch tatsächlich kritisch ist.

## 10.3 Die Problemlösung

Dieser Abschnitt erläutert, wie das vorliegende Problem modelliert werden kann und welche Distribuierungsstrategien erfolgreich waren und welche nicht. Insbesondere die Suche nach einer erfolgreichen Distribuierungsstrategie erwies sich als schwierig.

### 10.3.1 Das Modell

Der Startzeitpunkt einer Prozeßausführung oder einer Nachrichtenübertragung wird durch eine Variable mit geeignetem Bereich modelliert (siehe Abschnitt 10.4). Auch diese Variablen werden jeweils mit $start(P_i)$ und $start(M_j)$ bezeichnet. Da die Zykluszeit $CT$ typischerweise nur den Bruchteil einer Sekunde beträgt, muß der anfängliche Bereich der Startzeitvariablen sorgfältig gewählt werden. Die kleinste relevante Zeiteinheit ist der Kehrwert der Frequenz $\rho$ der globalen Uhr. Die Frequenz der globalen Uhr stimmt mit der des Datenbuses überein. Wir können keine größeren Zeiteinheiten wählen, da diese feine Auflösung für einige problemspezifische Konstanten verlangt wird. Somit werden alle Zeitlängen in Längeneinheiten $\frac{1}{\rho}$ konvertiert. Für die Startzeiten ergeben sich dann die folgenden Bereichsconstraints

$$\begin{aligned} start(P_i) &\in \{0,\ldots,CT - dur(P)\}, \\ start(M_j) &\in \{0,\ldots,CT - dur(M)\}. \end{aligned}$$

Diese Bereichsconstraints garantieren, daß weder $compl(P_i)$ noch $compl(M_j)$ größer werden kann als $CT$. Somit sind die Constraints (10.4) und (10.8) erfüllt. Um die Zahl der Variablen klein zu halten, werden die Endzeiten jedoch nicht explizit modelliert, sondern es werden Ausdrücke wie $start(P_i) + dur(P)$ oder $start(M_j) + dur(M)$ verwendet. Die übrigen Constraints (10.2), (10.5), (10.7) sowie (10.9)–(10.11) können durch Projektoren für Gleichungen und Ungleichungen realisiert werden wie sie in Abschnitt 4.1 beschrieben wurden. Die Ausnahme sind Kapazitäts- und Latenzconstraints.

**Kapazitätsconstraints** Die Kapazitätsconstraints (10.5) und (10.10) können durch reifizierte Constraints modelliert werden (siehe Kapitel 5 und 4.4). Constraint (10.5) kann zum Beispiel durch

$$\begin{gathered} (start(P_i) + dur(P) \leq start(Q_j)) \leftrightarrow B_1 = 1 \\ (start(Q_j) + dur(Q) \leq start(P_i)) \leftrightarrow B_2 = 1 \\ B_1 + B_2 = 1 \end{gathered}$$

modelliert werden.

Mit diesem Ansatz werden insgesamt $3 \cdot \frac{n \cdot (n-1)}{2}$ Constraints benötigt, um $n$ Prozeßausführungen oder Nachrichtenübertragungen zu serialisieren (siehe auch Abschnitt 5.2.1). Ein solcher Anzatz funktioniert zwar, solange $n$ relativ klein ist. Aber allein für das hier betrachtete Problem sind mehr als 1 800 Nachrichten zu übertragen, so daß dieser Ansatz nicht mehr realistisch ist. Auch

wenn viele subsumierte reifizierte Constraints in einer konkreten Implementierung dann durch die Speicherbereinigung beseitigt werden können, existiert eine sehr große Menge für relativ lange Zeit im Speicher.

Ein solch übermäßiger Speicherverbrauch kann durch die Verwendung von globalen Constraints vermieden werden (siehe Abschnitt 4.2). Dann stellt sich aber die Frage nach der Stärke der Propagierung, das heißt, welche Projektoren ausgewählt werden sollen. Wir haben in Abschnitt 5.2.1 einen Projektor betrachtet, der genau das Propagierungsverhalten der quadratischen Anzahl reifizierter Constraints von oben realisiert. Dieser Projektor spart in einer Implementierung nicht nur Speicher, sondern auch Laufzeit, da nur ein einzelner Projektor ausgeführt und verwaltet werden muß.

Eine Alternative ist die Verwendung von Edge-Finding wie es in Abschnitt 5.2.2 eingeführt wurde und für mehrere Projektoren mit unterschiedlicher Propagierungsstärke verwendet wurde. Wie in Kapitel 9 beschrieben, ermöglicht Edge-Finding das Lösen harter Schedulingprobleme. Edge-Finding ist zwar von der Berechnung her aufwendiger, kann aber den Suchraum möglicherweise drastisch reduzieren. Die Verwendung von Edge-Finding für das vorliegende Problem lohnt sich jedoch nicht. Der resultierende Suchbaum ist vielmehr identisch zu demjenigen bei Verwendung des Projektors aus Abschnitt 5.2.1. Dies liegt daran, daß die Bereiche der möglichen Startzeitvariablen im Vergleich zu den jeweiligen Ausführungs- bzw. Übertragungszeiten nicht sehr stark eingeschränkt sind. Relativ kleine Bereiche sind aber eine Voraussetzung dafür, daß Edge-Finding viel propagieren kann. Obwohl der Datenbus der tatsächliche Engpaß des Problems ist, spiegelt sich dies nicht unmittelbar in einer Reduktion der Bereichsgrößen wider (siehe auch Abschnitt 10.5), so daß Edge-Finding hier nicht viel Propagierung beisteuern kann. Für die Prozesse kann Edge-Finding zwar den Suchraum etwas einschränken (wegen der größeren Auslastung im Vergleich zum Datenbus sind die Bereiche kleiner), aber dies hat keinen Einfluß auf die Effizienz der Lösungssuche. Somit erhöht sich lediglich die Laufzeit bei Verwendung des Projektors aus Abschnitt 5.2.4 um etwa den Faktor Fünf. Deshalb werden die Constraints (10.5) und (10.10) durch den Projektor für den einfachen globalen Constraint realisiert.

**Latenzconstraints** Die Latenzbedingung (10.11) unterscheidet zwischen Kommunikationsmustern, die auf einem oder auf zwei Wiederholungsfenstern zu finden sind. Die Bedingungen in (10.11) können mit Hilfe eines reifizierten Constraints modelliert werden. Die Realisierung dieser Modellierung führt aber zu so wenig Constraintpropagierung, daß das Problem nicht innerhalb eines Tages Rechenzeit gelöst werden kann. Die Propagierung kann jedoch durch eine spezielle Vorverarbeitungsphase verbessert werden. Wenn zum Beispiel $P_i$ eine Nachricht an $Q_j$ sendet, kann vor der Distribuierungsphase festgelegt werden, ob $Q_j$ auf dem gleichen Wiederholungsfenster wie $P_i$ plaziert werden soll oder nicht. Ist eine solche Kommunikation nicht Teil eines Zyklus, wird angenommen, daß $Q_j$ nicht verzögert wird und $start(Q_j) \geq compl(P_i)$ gelten muß. Ist die Kommunikation zwischen $P_i$ und $Q_j$ jedoch Teil einer zyklischen Kommunikation (ein Beispiel ist in Abbildung 10.4 gezeigt), kann die Ordnung zwischen $P_i$ und $Q_j$ beliebig gewählt werden (in den vorliegenden Problemen gibt es jedoch nur Zyklen, bei denen der Sender und der Empfänger den gleichen Prozeß bezeichnen). Damit muß höchstens einer der folgenden Constraints durch einen Projektor realisiert werden.

$$\begin{aligned} start(Q_j) + dur(Q) - start(P_i) &\leq latency(M,Q), \\ CT - start(P_i) + start(Q_j) + dur(Q) &\leq latency(M,Q). \end{aligned}$$

Dies garantiert korrekte Propagierung. Durch diese Vorverarbeitung könnten jedoch alle möglichen Lösungen verlorengehen oder das Problem sehr viel schwieriger werden. Für die betrachteten Probleme ist dieses Vorgehen aber erfolgreich und daher gerechtfertigt (siehe auch Abschnitt 10.7).

### 10.3.2 Die Distribuierungsstrategie

Übliche Distribuierungsstrategien wie first-fail sind bei weitem zu schwach, um auch nur kleine Probleminstanzen zu lösen. Deshalb wurde eine speziell für Echtzeitanwendungen entwickelte Strategie und Strategien aus dem Operations Research (die Serialisierer aus Kapitel 6) auf ihre Eignung hin untersucht.

**Eine Strategie für Echtzeitanwendungen** Die zuerst untersuchte Distribuierungsstrategie wurde durch die Arbeit von Liu und Layland [LL73] motiviert. Diese beschäftigten sich mit Schedulingalgorithmen für Echtzeitanwendungen. Die Strategie arbeitet wie folgt. Sei $lct(P_i)$ bzw. $lct(M_i)$ das spätest mögliche Ausführungs- bzw. Übertragungsende von $P_i$ bzw. $M_i$. Der Wert $lct(P_i)$ beispielsweise ist der größte Wert des Bereichs von $P_i$ plus seine Übertragungsdauer. Es wird dann genau dasjenige Paar $(P_i, M_i)$ ausgewählt, für das $(lct(P_i), lct(M_i))$ minimal bezüglich der lexikographischen Ordnung ist. Außerdem kann ein Paar nur dann ausgewählt werden, wenn entweder $start(P_i)$ oder $start(M_i)$ noch nicht determiniert ist. Wenn $start(P_i)$ noch nicht determiniert ist, wird mit $\{start(P_i) = c\}$ distribuiert, wobei mit der kleinsten Zahl in dem Bereich von $P_i$ als Wert für $c$ begonnen wird. Sobald ein konsistenter Wert für $start(P_i)$ gefunden ist, wird der Startzeitpunkt für $M_i$ auf die gleiche Weise determiniert. Somit bestimmt diejenige Prozeßausführung mit der kleinsten spätest möglichen Endzeit (üblicherweise ein Prozeß mit einer hohen Frequenz), welche Startzeitpunkte zuerst determiniert werden. Es sind dies die Startzeiten all derjenigen Nachrichtenausführungen, die von genau dieser Prozeßausführung gesendet werden. Ist nur $start(M_i)$ noch nicht determininert, so wird wie für $P_i$ verfahren. Obwohl diese Strategie gut funktioniert, wenn keine Latenzconstraints betrachtet werden, wird bei Berücksichtigung von Latenzconstraints selbst nach einem Tag Rechenzeit keine Lösung gefunden.

**Strategien aus dem Operations Research** Alternativ wurden verschiedene Serialisierer getestet. Die zugehörige Distribuierungsstrategie kann in zwei Phasen aufgeteilt werden. In der ersten Phase wird eine passende Serialisierung gefunden, während in der zweiten Phase konkrete Startzeiten festgelegt werden (siehe Kapitel 6). Dabei bedeutet *Serialisierung*, daß für alle $s, s' \in S$ mit $s \neq s'$ bestimmt werden muß, ob $s$ vor $s'$ plaziert wird oder umgekehrt. Eine entsprechende totale Ordnung von $S$ kann man erhalten, indem mit einem der folgenden Constraints distribuiert wird.

$$start(s) + dur(s) \leq start(s') \quad \text{oder} \quad start(s') + dur(s') \leq start(s).$$

Im besten Fall werden $\frac{|S| \cdot |S-1|}{2}$ Distribuierungsschritte benötigt, um $S$ vollständig zu serialisieren. Diese Zahl kann reduziert werden, indem einige Ordnungsentscheidungen deterministisch festgelegt werden. Zum Beispiel vermeidet die in [CL95] vorgestellte Strategie Distribuierungsschritte durch das Bestimmen von Ordnungen, die notwendigerweise gelten müssen (durch eine dem

Edge-Finding ähnliche Technik; siehe auch Abschnitt 6.3). In Abschnitt 9.4 wurde gezeigt, daß diese Strategien exzellente Resultate liefern können, ohne daß die theoretisch mögliche Tiefe des Suchbaumes erreicht wird. Dies gilt aber nur für den Fall, daß die Bereiche der Variablen relativ klein sind (wie zum Beispiel beim Optimalitätsbeweis). Soll jedoch einfach eine Lösung gefunden werden, so zeigen sich die Nachteile dieser Strategien, da die Tiefe des Suchbaumes drastisch anwächst. Weil – wie in Abschnitt 10.3.1 bereits gezeigt – Edge-Finding keinen Nutzen für die betrachteten Probleme bringt, kann auch die Strategie aus [CL95] nicht viele Ordnungsentscheidungen deterministisch treffen (eine Ausnahme sind die Entscheidungen, die für die Reihenfolge jeweils zweier Instanzen eines Prozesses bzw. einer Nachricht getroffen werden können). Eine Strategie, für die die Zahl der notwendigen Distribuierungsschritte quadratisch in der Größe von $S$ wächst, ist jedoch nicht realistisch für Probleme der hier betrachteten Größenordnung. Zudem wächst für Aufgabenintervalle die Laufzeit kubisch mit der Anzahl der Aufgaben. Auch dies läßt aufgabenorientierte Strategien wie die aus [CL95] ausscheiden.

Trotzdem haben wir die Strategien aus [CL95] und [SC93] auf ihre prinzipielle Eignung hin getestet. Leider konnte mit keiner der beiden Strategien nach mehreren Stunden Berechnungszeit eine Lösung selbst für ein kleines Testbeispiel gefunden werden (das Testbeispiel war um mehr als eine Größenordnung kleiner als die realen Probleme). Damit sind diese Strategien, die mit paarweisen Ordnungsentscheidungen distribuieren, sogar zu schwach für die betrachteten Probleme. Die schlechte Qualität dieser Strategien liegt (wie für Edge-Finding) an den anfänglich nur gering eingeschränkten Bereichen (das Verhältnis von Bereichsbreite zur Ausführungs- bzw. Übertragungsdauer ist sehr groß), so daß die Ordnungsentscheidungen nicht gut fundiert sind. Damit kann die Strategie die Suche in eine falsche Richtung lenken, die bei einfacher Tiefensuche nicht einfach wieder umgekehrt werden kann.

Anstatt nur ein Paar von Aufgaben pro Distribuierungsschritt zu ordnen, können auch mehrere Paare in einem Schritt geordnet werden. Die Anzahl der Schritte kann dadurch im günstigen Fall drastisch verringert werden (siehe auch Abschnitt 6.2 zu ressourcenorientierten Serialisierern). Zuerst wird ein Element $s \in S$ ausgewählt. Dann wird mit den Constraints distribuiert, daß $s$ vor allen Elementen in $S$ (außer $s$ selbst) plaziert wird:

$$start(s) + dur(s) \leq start(s'), \text{ für alle } s' \in S \text{ , so daß } s' \neq s.$$

Führt dies zu einem Fehlschlagen, wird dieses Vorgehen mit einem anderen $s \in S$ wiederholt. Ansonsten wird $s$ aus $S$ entfernt. Sobald die nachfolgende Propagierung einen Fixpunkt erreicht hat, wird wieder ein Distribuierungsschritt ausgeführt. Dies geschieht im Wechsel bis alle Elemente geordnet sind. Im besten Fall werden nicht mehr als $|S| - 1$ Distribuierungsschritte zur Serialisierung von $S$ benötigt. Natürlich müssen nach wie vor quadratisch viele Ordnungsentscheidungen getroffen werden, aber die Tiefe des Suchbaumes kann möglicherweise reduziert werden.

Es bleibt noch die Frage zu beantworten, welche der möglichen Kandidaten für $s \in S$ zuerst getestet werden sollen. Wie in Abschnitt 6.2 beschrieben, können in linearer Zeit, also $O(|S|)$, die Elemente in $S$ bestimmt werden, die vor allen übrigen plaziert werden können [CP89, BPN95a, Wür96]. Sei $F$ diese Menge. Ist $F$ leer, kann keine Serialisierung von $S$ existieren. Gilt $|F| = 1$, so ist kein Distribuierungsschritt notwendig, da das einzige Element in $F$ der einzige Kandidat für $S$ ist. Für die betrachteten Probleme ist dies für den Datenbus kaum, für die Prozessoren jedoch häufiger der Fall. Da für die Kandidatenauslese eine dem Edge-Finding verwandte Technik verwendet wird und die Bereiche der Prozesse im Vergleich zu den Bereichen der Nachrichten

kleiner sind, ist diese Beobachtung erklärbar (siehe auch die Diskussion in den vorangehenden Abschnitten). Gilt andererseits $|F| > 1$, wird dasjenige Element in $F$ zuerst ausgewählt, dessen frühest mögliche Startzeit am kleinsten ist. Ist dieser Wert für zwei Elemente gleich, wird dasjenige Element mit der kleinsten spätest möglichen Startzeit ausgewählt (insgesamt wird also nach der lexikographischen Ordnung $(est(s), lst(s))$ ausgewählt). Dies garantiert, daß für die übrigen Elemente so viel Platz wie möglich zum Verschieben übrigbleibt.

**Reihenfolge der Serialisierung** Nun muß noch entschieden werden, welche Ressource zuerst serialisert werden muß; die Prozessoren oder der Datenbus. Aus den vorangehenden Kapiteln wissen wir, daß möglichst ein Engpaß zuerst serialisiert werden sollte. Als ein übliches Kriterium wurde die jeweilige Auslastung gewählt (also der Quotient aus der insgesamt verfügbaren Zeit und der benötigten Ausführungsdauer bzw. Übertragungszeit; vergleiche auch das Schlupf-Kriterium in Kapitel 6). Da die Auslastung der Prozessoren bis zu 92% beträgt, die des Datenbuses aber nur etwa 11%, sollten die Prozessoren vor dem Datenbus serialisiert werden. Überraschenderweise konnten damit nur wenige aber nicht alle der vorliegenden Probleme gelöst werden (nur zwei von sieben). Analysiert man das Problem genauer, so stellt sich heraus daß der Ursprung der Komplexität das Zusammenwirken von periodischen Prozessen und Nachrichten ist (siehe auch Abschnitt 10.5). Prozesse auf verschiedenen Prozessoren mit unterschiedlichen Perioden interagieren über den gemeinsamen Datenbus durch Nachrichtenübertragungen. Die Constraints für die über den Bus übertragenen Nachrichten führen zu Abhängigkeiten zwischen nicht direkt kommunizierenden Prozessen. Somit wurde die oben beschriebene Distribuierungsstrategie zuerst zur Serialisierung der Nachrichtenausführungen auf dem Datenbus angewendet. Die Serialisierung der Nachrichten führt automatisch zu einer Reduktion der möglichen Startzeiten der sendenden Prozesse. Anschließend wurden die Prozeßausführungen auf die gleiche Weise serialisiert. In der aktuellen Implementierung ist es nicht notwendig zu spezifizieren, welche Prozessoren zuerst serialisiert werden (siehe aber Abschnitt 10.7).

**Bestimmen einer konkreten Startzeit** Zum Bestimmen konkreter Startzeiten ist es ausreichend, nach der Serialisierung die Startzeiten auf den kleinsten noch möglichen Wert festzulegen. Dadurch ist kein weiterer Distribuierungsschritt mehr notwendig. Diese Tatsache kann von einem Satz von Van Hentenryck [VD91a] abgeleitet werden. Dieser Satz besagt, daß Constraintpropagierung alleine zum Entdecken von Inkonsistenz ausreicht, wenn die Constraints alle von der Form $x + c \leq y$ und $x + c = y$ sind, wobei $c$ eine ganze Zahl ist. Der Satz besagt weiterhin, daß eine Lösung auf sehr ähnliche Art wie eben beschrieben erhalten werden kann. Van Hentenrycks Satz enthält aber keine disjunktiven Constraints wie $x + c_1 \leq y \vee y + c_2 \leq x$, die im hier betrachteten Problem vorkommen. Da aber durch die Serialisierung für jeden disjunktiven Constraint eine der beiden Klauseln realisiert wird, kann der Satz auch für die hier betrachteten Probleme angewendet werden.

## 10.4 Empirische Ergebnisse

Das in Abschnitt 10.3 beschriebene Problem wurde in DFKI Oz, Version 2.0.4, realisiert und mit mehreren großen Problemen aus dem Daimler-Benz-Konzern getestet (aktuell sind sieben

Probleme verfügbar). Die Frequenz der globalen Uhr führt zu möglichen Startzeiten zwischen Null und 6 Millionen. Es gibt 20 Prozessoren, die insgesamt etwa 170 Prozesse beherbergen. Die Prozesse führen zu etwa 350 Prozeßausführungen. Die Prozeßausführungen übertragen um die 1 200 Inter-Prozessor-Nachrichten. Mit zusätzlich 600 Resynchronisationsnachrichten sind mehr als 1 800 Nachrichten über den Datenbus zu übertragen. Dies ist gleichzeitig die größte Zahl von Argumenten eines Kapazitätsconstraints und eines Serialisierers. Mit den Resynchronisationsnachrichten gibt es mehr als 1 200 Nachrichten mit einer höheren Frequenz als die des Wiederholungsfensters. Es gibt etwa 150 Latenzconstraints für unterschiedliche Nachrichten und kommunizierende Prozesse. Die gesamte Auslastung des Datenbuses beträgt etwa 11% und die maximale Auslastung eines Prozessors etwa 92%.

Das Programm berechnet eine erste Lösung auf einer Sun Ultra-1 mit 170 MHz in etwa 10 Minuten mit einem aktiven Speicherverbrauch auf der Halde von rund 7.5 MB. Im Laufe der Berechnung werden mehr als 1.6 Millionen Propagierer erzeugt, von denen etwa 23 000 vor dem ersten Distribuierungsschritt existierten (für das schwierigste Problem werden sogar 3.9 Millionen Propagierer erzeugt). Für die betrachteten Probleme liegt die Anzahl der Fehlerknoten im Suchbaum zwischen Null und 40, ist also sehr klein. Die Anzahl der Distribuierungsschritte liegt bei mindestens 2 000, die resultierenden Suchbäume sind also sehr tief.

Die Zahl der erzeugten Propagierer steht nicht im Widerspruch zu dem erwähnten Speicherverbrauch. Viele der während der Serialisierung erzeugten Propagierer sind früh subsumiert und werden deshalb durch die Speicherbereinigung beseitigt. Es ist hier wichtig, daß Subsumtion der Speicherbereinigung nicht erst signalisiert wird, wenn alle Variablen determiniert sind (wie es zum Beispiel in `clp(FD)` [CD96] und in AKL(FD) [CCJ95] der Fall ist). Dies würde in unserem Fall zu spät sein, da erst nach der Serialisierung des Datenbuses und der Prozessoren die Startzeiten determiniert werden.

Um den Speicherverbrauch gering zu halten, wurde das Konzept der *Wiederberechnung* (engl. *recomputuation*) von Berechnungsräumen in Oz ausgenutzt [Sch98b]. Distribuierung wird in Oz standardmäßig durch das Kopieren von Berechnungsräumen erreicht (siehe Abschnitt 7.1.4). Dies kann durch Wiederberechnung vermieden werden. Dabei wird eine Wiederberechnungstiefe $t$ spezifiziert, die angibt, nach wieviel Verzweigungen im Suchbaum eine Kopie gemacht werden soll. Ansonsten wird bei einer Wahlanweisung nur vermerkt, welche Alternative der Wahlanweisung ausgewählt wurde. Schlägt ein Berechnungsraum fehl, so wird auf die zuletzt gemachte Kopie zurückgesetzt und der die Wahlanweisung enthaltene Berechnungsraum wird durch Wiederberechnung rekonstruiert. Der Wert von $t$ für diese Anwendung beträgt 800. Wiederberechnung ist essentiell zur Lösung des Problems.

In [CA95] wird ein Problem ähnlicher Größenordnung gelöst. Hierzu ist es aber notwendig, die Periodizitätsconstraints teilweise zu verletzen, da der gleiche Abstand zwischen Instanzen eines Prozesses nicht eingehalten werden kann. Es wird in [CA95] nicht klar, inwieweit dies durch das Problem oder durch das verwendete Lösungsverfahren bedingt ist.

## 10.5 Die Komplexität des Problems

Schedulingprobleme sind typischerweise NP-vollständig. Die NP-Vollständigkeit der hier betrachteten Probleme folgt jedoch nicht unmittelbar aus den in [GJ79, Seiten 236–244] aufgeführ-

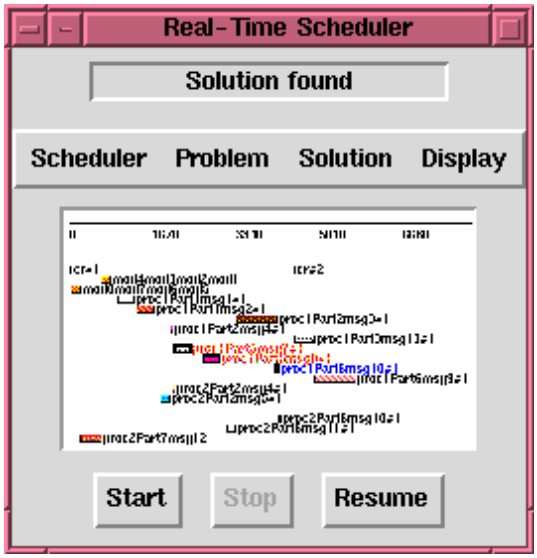
ten Resultaten. Sieht man das vorliegende Problem als Job-Shop-Problem (siehe Abschnitt 9.3.3), so besteht hier ein Job aus nur zwei Aufgaben, nämlich einem Sender und einer Nachricht. Die zweite Aufgabe wird sogar immer auf der gleichen Ressource plaziert. Es ist jedoch bekannt, daß Job-Shop-Probleme in polynomialer Rechenzeit gelöst werden können, wenn es nur zwei Ressourcen gibt und die zweite Aufgabe eines Jobs nicht immer auf der gleichen Ressource ausgeführt werden muß [Jac56]. Somit stellt sich die Frage, ob das hier betrachtete Problem trotzdem NP-vollständig ist. Durch eine Reduktion des sogenannten Bin-Packing-Problems auf das hier vorgestellte Problem kann tatsächlich NP-Vollständigkeit gezeigt werden. Ein ausführlicher Beweis ist in [SW97] zu finden.

Der Beweis der NP-Vollständigkeit macht deutlich, daß die primäre Komplexitätsquelle die verschiedenen Perioden der Prozesse sind. Denn der Beweis kann ohne zusätzliche Constraints nur mit den Kapazitäts- und Periodizitätsconstraints geführt werden. Die Interaktion aller Prozesse und ihrer Inter-Prozessor-Nachrichten auf dem gemeinsamen Datenbus ist jedoch eine weitere Komplexitätsquelle (wie auch Latenz- und Präzedenzconstraints), die die praktische Lösung des Schedulingproblems für den Datenbus so schwierig macht.

## 10.6 Der Prototyp

Im Rahmen des Projekts wurde ein Prototyp zur Lösung des vorgestellten Schedulingproblems implementiert (siehe Abbildung 10.5). Dieser Prototyp verwendet auch Techniken, die für den Oz-Scheduler entwickelt wurden (wie zum Beispiel den Scheduling-Compiler; siehe Kapitel 9).

**Abbildung 10.5** Prototyp




Die Funktionalität des Prototyps kann über entsprechende Menüs bzw. Knöpfe ausgewählt werden. Ausgehend von der vom Kunden gelieferten Originalproblembeschreibung kann eine Oz-Datenstruktur erzeugt werden, die alle wichtigen Konstanten des Problems (wie Uhrenfrequenz etc.) und die vorkommenden Prozessoren, Nachrichten und Prozesse enthält. Diese Datenstruktur dient dann als Problemspezifikation. Mittels verschiedener Knöpfe kann die Lösungssuche ge-

startet, unterbrochen oder wiederaufgenommen werden. Statistische Daten zu dem Problem wie zum Beispiel die Frequenz der Resynchronisationsnachrichten, die Anzahl der Prozeßausführungen, Datenbus- bzw. Prozessorauslastung usw. können abgerufen werden. Ein Gantt-Diagramm visualisiert die generierte Lösung, und durch Klicken im Diagramm mit der Maus kann der Benutzer Informationen über die ausgewählte Prozeßausführung bzw. Nachrichtenübertragung im Oz-Browser [HMSW97] inspizieren. Zusätzlich wird das Kommunikationsmuster (Sender, Nachricht, Empfänger) durch farbige Unterlegung verdeutlicht. Die gefundene Lösung kann in einer passenden Datenstruktur in einer Datei abgespeichert werden. Auch von einer abgespeicherten Lösung kann ein Gantt-Diagramm erstellt werden.

## 10.7 Ausblick

In einer Vorstudie wurde die Problemgröße nochmals gesteigert und mehr Typen von Prozessen und Nachrichtenübertragungen zugelassen. Dies resultierte in Problemen mit bis zu 500 Prozeßausführungen und mehr als 3 000 Nachrichtenübertragungen. Die besten Resultate wurden erzielt, indem in Oz als Distribuierer `FD.schedule.firstsLastsDist` verwendet wurde (siehe [HMSW97] und Seite 95). Außerdem wurden Prozesse auf Prozessoren, die mit Ein- und Ausgabe behaftet sind, zuerst serialisiert.

Für ein Anschlußprojekt kann es eventuell nötig werden, die Vorverarbeitungsphase für zyklische Abhängigkeiten durch ein dynamisches Vorgehen zu ersetzen (falls das Problem durch diese Phase dann zu stark eingeschränkt wird). Dies kann zum Beispiel dadurch realisiert werden, indem die Kommunikationsmuster dynamisch in Abhängigkeit von den aktuellen Bereichen bestimmt werden. Ein solches Vorgehen würde dann mit der Serialisierung eng verzahnt sein.

# Kapitel 11

# Implementierung

In diesem Kapitel wird die Implementierung des FD-Systems von Oz durch die sogenannte *abstrakte Maschine* beschrieben. Die abstrakte Maschine interpretiert zur Laufzeit die *abstrakten Maschinenbefehle*, die das Ergebnis der Übersetzung eines Oz-Programms sind.

Wir konzentrieren uns in dieser Arbeit auf die Implementierung von FD-Variablen und Propagierern unter besonderer Berücksichtigung von Berechnungsräumen. Wir werden hier jedoch nicht die exakte Übersetzung in Maschinenbefehle erläutern, sondern die prinzipielle Integration von FD-Variablen und Propagierern in die abstrakte Maschine beschreiben. Für die Implementierung weiterer Sprachkonstrukte von Oz sei auf [MSS95, Meh98, Sch98a, Sch98b, MW95, WM96] verwiesen.

Dieses Kapitel enthält die erste ausführliche Darstellung der Implementierung des gewählten Propagierungsmodells in einer nebenläufigen Constraintsprache. Wir zeigen insbesondere zum ersten Mal, wie nicht-monotone Propagierungsfunktionen integriert werden können, wie beliebige Propagierer reifiziert werden können und wie aus einer Hochsprache mit Propagierern kommuniziert werden kann. Die beschriebene Implementierung von Propagierern kann jedoch auch leicht zur Realisierung von Systemen spezialisiert werden, die keine Berechnungsräume anbieten.

Abschnitt 11.1 gibt einen Überblick über die Integration von Propagierern in die abstrakte Maschine und deren Funktionsweise. Die Abschnitte 11.2 und 11.3 beschreiben die Implementierung von FD-Variablen und der Basisconstraints unter besonderer Berücksichtigung von Berechnungsräumen. Abschnitt 11.4 enthält die Realisierung von Propagierern und wie diese in die abstrakte Maschine integriert werden. Die beiden folgenden Abschnitte befassen sich mit Optimierungen und mit Fairneß. In Abschnitt 11.7 wird u. a. beschrieben, wie selbst nicht-monotone Algorithmen zur Propagiererrealisation genutzt werden können, wie man Propagierer über ein Oz-Programm steuern kann oder wie reifizierte Constraints durch spekulative Berechnung in Propagierern implementiert werden können. Das Kapitel endet mit einem Implementierungsausblick.

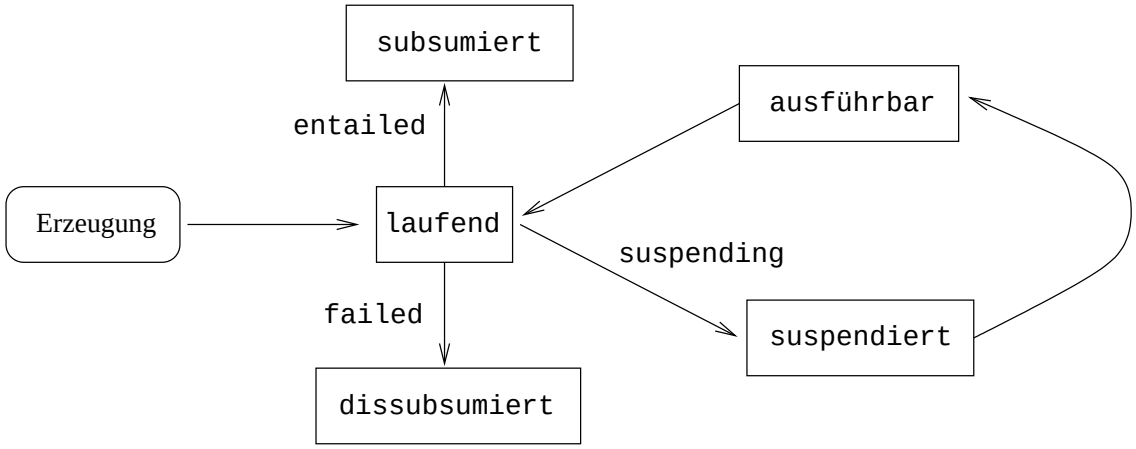
In diesem Kapitel werden wir oft im Text die Implementierung von Oz mit der Implementierung von `clp(FD)` und AKL(FD) vergleichen, da diese Systeme am genauesten dokumentiert sind. Über die Implementierung von Systemen wie CHIP oder ILOG SOLVER ist jedoch nur wenig

bekannt.

## 11.1 Überblick

In der abstrakten Maschine ist ein Propagierer als eine Prozedur implementiert. Diese Prozedur realisiert die Propagierungsfunktion $N_p$ und die Subsumtionsfunktion $E_p$ eines Projektors $p$ aus Abschnitt 2.4, der einen Propagierer definiert. Die abstrakte Maschine führt den Propagierer (also die entsprechende Prozedur) aus und erhält vom Propagierer Informationen, die für die Feststellung des Propagiererzustands verwendet werden. In der Implementierung kann ein Propagierer fünf verschiedene Zustände annehmen (siehe Abbildung 11.1). Nach der Erzeugung eines Propagierers durch eine vordefinierte Oz-Prozedur, wird der Propagierer ausgeführt (Zustand `laufend`). Je nach Ergebnis gibt der Propagierer `failed` zurück, falls er feststellt, daß der realisierte Constraint `dissubsumiert` ist, `entailed`, falls der Propagierer feststellt, daß der realisierte Constraint `subsumiert` ist, und `suspending`, falls keiner der beiden vorherigen Fälle gilt. Im letzten Fall ist der Propagierer im Zustand `suspendiert`. Wird später zum Beispiel der Bereich eines Arguments des Propagierers verkleinert, so wird der Propagierer ausführbar (siehe unten). Ist ein Propagierer im Zustand `subsumiert` oder `dissubsumiert`, wird der Propagierer von der abstrakten Maschine gelöscht. Im folgenden werden wir die den Propagierer implementierende Prozedur mit dem Propagierer selbst identifizieren.

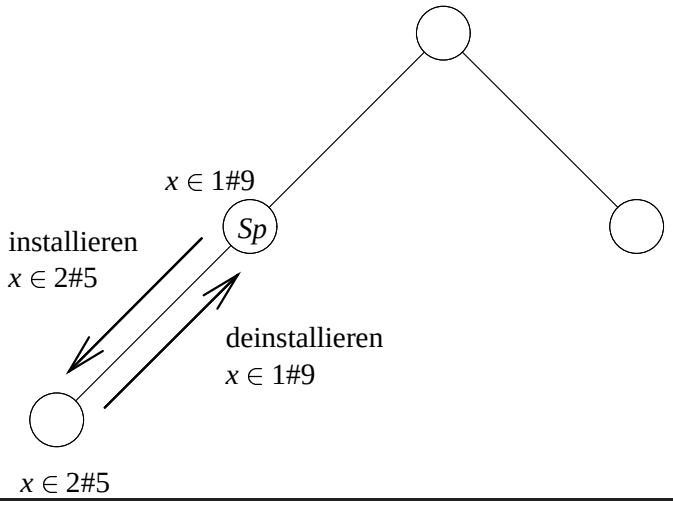
**Abbildung 11.1** Zustände eines Propagierers




Konzeptuell hat jeder Berechnungsraum einen eigenen Constraintspeicher (siehe Abschnitt 7.1.4). Dies ist in der Implementierung aus Effizienzgründen aber nicht der Fall. Stattdessen ist ein inkrementeller Ansatz implementiert, wobei es nur eine einzige Repräsentation für die Constraintspeicher gibt. Die abstrakte Maschine muß dann sicherstellen, daß bei Bewegungen in der Hierarchie der Berechnungsräume immer der zum gerade aktuellen Berechnungsraum gehörende Constraintspeicher verfügbar ist [MSS95]. Sei *Sp* ein Berechnungsraum, in dem die Variable $x$ deklariert wurde und für die der Bereichsconstraints $x \in 1\#9$ gilt (siehe Abbildung 11.2). Wird der Bereichsconstraint für $x$ nun in einem *Sp* untergeordnetem Berechnungsraum weiter zu $x \in 2\#5$ verstärkt, muß der alte Bereichsconstraint ($x \in 1\#9$) so vermerkt

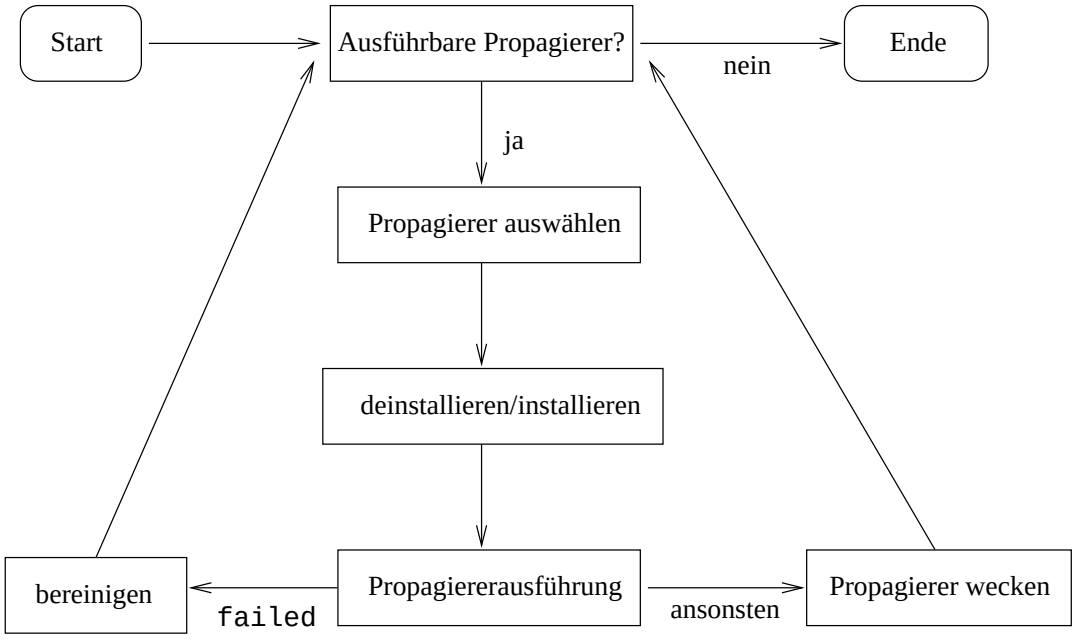
werden, daß bei Verlassen (*deinstallieren*) des untergeordneten Berechnungsraumes der stärkere Constraint wieder zurückgenommen werden kann. Umgekehrt muß beim Verlassen eines Berechnungsraumes der stärkere Bereichsconstraint ($x \in 2\#5$) vermerkt werden, um bei einem späteren Wiederbetreten (*installieren*) mit diesem stärkeren Constraint die Berechnung fortsetzen zu können.

**Abbildung 11.2** Deinstallation und Installation von Berechnungsräumen




Die Funktionsweise der abstrakten Maschine ist in Abbildung 11.3 gezeigt. Dabei ist insbesondere der Existenz einer Hierarchie von Berechnungsräumen Rechnung getragen. Wir nehmen hier an, daß nur Propagierer auszuführen sind (wir gehen auf die Reduktionsregeln für die übrigen Anweisungen in Abschnitt 7.1.3 also nicht ein; siehe hierfür [MSS95, Meh98, Sch98a, Sch98b]). In der abstrakten Maschine gibt es eine globale Datenstruktur, in der ausführbare Propagierer gespeichert werden (typischerweise eine Warteschlange). Ist kein Propagierer ausführbar, so kann die Berechnung beendet werden. Ansonsten wird ein Propagierer ausgewählt und es wird geprüft, ob der Propagierer in einem anderen als dem aktuellen Berechnungsraum ausgeführt werden muß. In diesem Fall wird der aktuelle Berechnungsraum deinstalliert und der Berechnungsraum des Propagierers installiert (für Details siehe Abschnitt 11.4.2). Danach entfernt die abstrakte Maschine den Propagierer aus der Warteschlange, die die ausführbaren Propagierer enthält und führt den Propagierer aus. Der Propagierer realisiert die Propagierungsfunktion $N_p$ des implementierten Projektors $p$, indem er Basisconstraints zum Constraintspeicher hinzufügt. Ist das Ergebnis des Propagierers `failed`, so werden die Datenstrukturen zur Deinstallation von Berechnungsräumen bereinigt und der aktuelle Berechnungsraum schlägt fehl. Ansonsten werden all diejenigen Propagierer *geweckt* (sie sind dann im Zustand `ausführbar`), für deren Argumente der ausgeführte Propagierer Basisconstraints zum Constraintspeicher hinzugefügt hat (für Details siehe Abschnitt 11.2). In beiden Fällen wird dann wieder geprüft, ob es ausführbare Propagierer gibt. Auf diese Weise wird die Normalform eines Constraintspeichers bzgl. der im zugehörigen Berechnungsraum enthaltenen Propagierer berechnet (siehe auch Abschnitt 7.2.3).

**Abbildung 11.3** Die Ausführung von Propagierern




## 11.2 FD-Variablen

Basisconstraints im Constraintspeicher werden durch Information repräsentiert, die an Variablen gebunden ist (sogenannte *variablenzentrierte* Repräsentation).

Eine FD-Variable hat die folgende Struktur (siehe auch Abbildung 11.4).

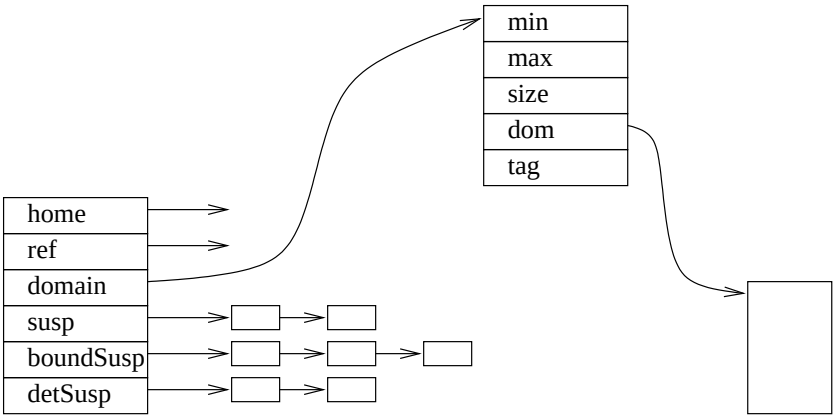
```
TYPE FDVar = RECORD
                home:      ↑Space;
                ref:       ↑FDVar;
                domain:    ↑FiniteDomain;
                susp:      ↑list of suspPtr;
                boundSusp: ↑list of suspPtr;
                detSusp:   ↑list of suspPtr;
             END
```

Die Datenstruktur für eine FD-Variable enthält einen Verweis auf den Berechnungsraum `home`, in dem die Variable deklariert wurde, einen Verweis auf eine weitere FD-Variable `ref`, falls die Variable gebunden wird (initial `nil`; siehe Abschnitt 11.3), einen Verweis auf eine Datenstruktur, die einen Bereich definiert (siehe Abschnitt 11.5.1 weshalb dies ein Verweis ist) und insgesamt drei Verweise auf sogenannte *Suspensionslisten*, die für das Wiederausführen von Propagierern benutzt werden (siehe unten).

Ein Bereich wird durch einen Verbund beschrieben, der die untere und obere Grenze sowie die Größe des Bereichs enthält (siehe auch Abbildung 11.4). In DFKI Oz ist die kleinstmögliche untere Grenze 0 und die größtmögliche obere Grenze $2^{27} - 2$, also 134 217 726.[1] Der Bereich selbst

[1] In DFKI Oz rechnen Propagierer aus Effizienzgründen mit sogenannten `smallInts`, für die 28 Bits zur

**Abbildung 11.4** Repräsentation einer FD-Variablen




ist entweder ein Intervall (`INTV`), ein Bitvektor (`BV`) oder eine Liste von Intervallen (`LIST`). Enthält ein Bereich Lücken, so wird ein vom Benutzer einstellbarer Wert `maxBV` dazu verwendet, zwischen einer Repräsentation durch einen Bitvektor oder einer Intervalliste zu entscheiden. Ist die obere Grenze des Bereichs kleiner als `maxBV`, wird ein Bitvektor verwendet, anderenfalls wird eine Intervalliste verwendet (aus implementierungsinternen Gründen ist der vom System initial gesetzte Wert von `maxBV` gleich 288). Mittels des Typfeldes (`tag`) kann erkannt werden, von welchem Typ der Bereich ist. Auf den Bereich (oder `nil` im Falle `INTV`) wird durch das Feld `dom` verwiesen. Wir lassen keine negativen Zahlen zu, da dies in den von uns betrachteten Anwendungen nicht notwendig war.

In ECL$^i$PS$^e$ [ECR96] wird ein Bereich durch eine Liste von Intervallen, in `clp(FD)` [CD96] durch ein einzelnes Intervall oder einen Bitvektor und in AKL(FD) [Car95] durch ein einzelnes Intervall oder eine Liste von Bitvektoren repräsentiert.

```
TYPE FiniteDomain = RECORD
                       min:  integer;
                       max:  integer;
                       size: integer;
                       tag:  (INTV, BV, LIST);
                       dom:  ↑Extension;
                    END
```

Damit wird für $x \in$ 3#1000 keine zusätzliche Repräsentation des Bereichs benötigt, für $x \in$ [3#10 20#50] wird ein Bitvektor und für $x \in$ [1#5 1000#1010] eine Intervalliste verwendet. In dieser Arbeit nehmen wir an, daß eine natürliche Zahl durch eine entsprechende FD-Variable repräsentiert wird. In DFKI Oz werden für natürliche Zahlen jedoch eigene Datenstrukturen zur Verfügung gestellt.

Die Suspensionslisten enthalten Verweise auf geeignete *Suspensionseinträge*, um Propagierer wieder auszuführen (zu *wecken*), die den Constraintspeicher verstärken können (der Eintrag wird in Abschnitt 11.4 beschrieben). Das Durchgehen durch diese Suspensionslisten, um Propagierer zu wecken, bezeichnen wir als *Inspizieren*. Die drei Suspensionslisten unterscheiden

Verfügung stehen. Da auch negative Zahlen mit `smallInts` dargestellt werden müssen, ergibt sich letztendlich $2^{27} - 2$ als obere Grenze.

sich in dem Ereignis, das das Wecken der in ihr enthaltenen Propagierer auslöst. Es ist zum Beispiel nutzlos, einen nur mit den Grenzen von Bereichen rechnenden Propagierer (wie für $x < y$) aufzuwecken, wenn ein Wert aus der Mitte eines Bereichs entfernt wird. Die Liste `susp` enthält die Verweise für Propagierer, die bei jeder Bereichsveränderung eines ihrer Argumente zu wecken sind (z. B. `FD.atmost`, der besagt, daß genau $y$ Elemente in einer Liste von $x_1$ bis $x_n$ gleich der ganzen Zahl $v$ sein sollen). Die Liste `boundSusp` enthält die Verweise für Propagierer, die bei einer Änderung einer Bereichsgrenze wieder zu wecken sind (z. B. `X <: Y`), und `detSusp` diejenigen Verweise für Propagierer, die bei Determinierung einer FD-Variablen zu wecken sind (z. B. `FD.distinct`, der besagt, daß $n$ Variablen paarweise verschiedene Werte annehmen sollen). Das Konzept verschiedener Suspensionslisten ist bereits in `CHIP` verwirklicht worden [DVS+88]. Neben seinem Ergebnis (für die Aktualisierung des Propagiererzustands) teilt ein Propagierer der abstrakten Maschine auch mit, welche Argumente überhaupt geändert oder determiniert wurden und für welche Argumente die Grenzen der Bereiche verkleinert wurden. Dadurch kann die abstrakte Maschine die entsprechenden Suspensionslisten inspizieren.[2] Wird eine Variable determiniert, werden alle Suspensionslisten inspiziert. Ändert sich eine Bereichsgrenze, wird neben der Suspensionsliste `boundSusp` noch die Liste `susp` inspiziert.

Im Gegensatz zu `clp(FD)` oder AKL(FD) wird in Oz für die Suspensionslisten nicht zwischen Änderungen der unteren und oberen Schranke eines Bereichs unterschieden. Dies liegt daran, daß Propagierer wie für $x < y$ wegen des vorzunehmenden Subsumtionstests auch geweckt werden müssen, falls sich z. B. nur die obere Grenze von $x$ ändert. In `clp(FD)` und AKL(FD) werden jedoch keine Subsumtionstests gemacht.

## 11.3 Unifikation und Bereichsconstraints

In Abbildung 11.3 ist nur der Fall illustriert, daß alleine Propagierer auszuführen sind. Uns interessiert hier zusätzlich lediglich, wie Gleichheitsconstraints `x=y` und Bereichsconstraints implementiert sind.

Eine Variable, die in Anweisungen des Berechnungsraumes *Sp* vorkommt und in einem übergeordneten Berechnungsraum deklariert wurde, heißt *global* in *Sp*. Die in *Sp* deklarierten Variablen heißen *lokal*.

Ein Gleichheitsconstraint wird durch sogenannte *Unifikation* realisiert. Zuerst wird jede der beiden Variablen *dereferenziert*, also der Kette möglicher Verweise `ref` solange gefolgt, bis eine Struktur mit nicht gesetzter Referenz gefunden wird. Seien dies `x` und `y`. Variablen, die keine FD-Variablen sind, und deren Referenz nicht gesetzt ist heißen *ungebunden*. Ist `x` oder `y` eine ungebundene Variable, wird die ungebundene Variable an die andere (evtl. auch ungebundene) Variable *gebunden* (also `ref` entsprechend gesetzt). Ist die zu bindende Variable global, so wird im aktuellen Berechnungsraum vermerkt, daß diese Variable bei der Deinstallation dieses Raumes wieder zu einer ungebundenen Variable gemacht werden muß.

Wir betrachten jetzt den Fall, daß `x` und `y` beides FD-Variablen sind. Zuerst wird der Schnitt der zugehörigen Bereiche berechnet.[3] Ist der Schnitt leer, schlägt der Berechnungsraum fehl.

[2] In DFKI Oz ruft allerdings der Propagierer selbst ensprechende Weckfunktionen auf. Dies spart unnötige Kommunikation zwischen Propagierern und der abstrakten Maschine.

[3] Hierbei ist zu beachten, daß evtl. Bereiche unterschiedlicher Repräsentation geschnitten werden. Dabei kann es

Ansonsten wird unterschieden, ob die Variablen lokal oder global sind. Diese Fallunterscheidung ist nötig, da bei globalen Variablen evtl. Information für die Deinstallation vermerkt werden muß.

- x lokal und y lokal. Der Bereich von y wird zu dem Schnitt beider Bereiche geändert. Dann wird x an y gebunden.

- x global und y global. Aus dem Schnitt beider Bereiche wird eine neue lokale FD-Variable z erzeugt. Sowohl x als auch y werden an z gebunden. Für beide Variablen wird entsprechende Information für eine spätere Deinstallation des aktuellen Berechnungsraumes vermerkt. Durch die Bindung an z wird vermieden, daß x oder y bei evtl. weiteren Verstärkungen des Constraintspeichers mehrmals vermerkt werden. In `clp(FD)` und AKL(FD) werden sogenannte *timestamps* eingeführt, um zu verhindern, daß globale Variablen mehr als einmal für die Deinstallation vermerkt werden (dies erfordert dann einen weiteren Eintrag in der Variablenrepräsentation).

- x lokal und y global. Ist der Schnitt beider Bereiche echt kleiner als der Bereich von y, so wird y an x gebunden, wobei der Bereich von x auf den Schnitt geändert wird. Für y wird für die Deinstallation vermerkt, daß es gebunden wurde. Ist der Schnitt gleich dem Bereich von y, so wird x an y gebunden.

Ist eine FD-Variable an einer Unifikation beteiligt, so werden alle Suspensionslisten inspiziert und entsprechende Propagierer evtl. geweckt (siehe Abschnitt 11.4). Dies geschieht, da fast alle Propagierer auch die Gleichheit von Variablen zur Propagierung ausnutzen (sei es nur zu Vereinfachungen (siehe Abschnitt 11.4.1) oder zur Propagierung wie für `FD.distinct`, wo bei Gleichheit zweier Argumente des Propagierers Dissubsumtion festgestellt werden kann).

Ist x lokal und wird x an y gebunden, so gehen durch die Bindung von *x* alle Suspensionen von x verloren, die nicht zur Wiederausführung von Propagierern geführt haben (siehe Abschnitt 11.4.2). Deshalb müssen zuvor die Suspensionen von x zu y kopiert werden. Dieses Problem ist in [Car95] nicht beschrieben und wohl auch nicht erkannt worden.

Nun zu Bereichsconstraints wie $x \in \delta$. Nach der Dereferenzierung der Variablen zu x, wird geprüft, ob x eine ungebundene Variable ist. In diesem Fall wird eine lokale FD-Variable y mit dem $\delta$ entsprechenden Bereich erzeugt und x an y gebunden. Ansonsten wird der Schnitt des Bereichs von x und $\delta$ erzeugt. Ist dieser leer, schlägt der Berechnungsraum fehl. Ist x lokal, wird dessen Bereich zu dem Schnitt geändert. Ansonsten wird eine lokale Variable y mit dem Schnitt als Bereich erzeugt und x daran gebunden. Ist x global, wird in jedem Fall wieder Information zur Deinstallation des Berechnungsraumes vermerkt.

## 11.4 Propagierer

Ein Propagierer ist als C++-Prozedur implementiert, die bei der Ausführung des Propagierers durch die abstrakte Maschine aufgerufen wird. Die Argumente dieser Prozedur wurden aus dem

vorkommen, daß das Ergebnis einen abweichenden Typ hat. So hat der Schnitt eines Intervalls mit einer Intervalliste evtl. einen Bitvektor als Ergebnis.

entsprechenden Aufruf der vordefinierten Oz-Prozedur extrahiert. Durch die Ausführung der vordefinierten Oz-Prozedur wird die C++-Prozedur sofort ausgeführt (dies entspricht dem Erzeugen von Propagierern in Abschnitt 7.2.3). Je nach Ergebnis der Propagiererausführung schlägt der aktuelle Berechnungsraum fehl oder die Berechnung wird fortgesetzt.

Durch diese Implementierung ist die Ausführung eines Propagierers atomar. Im Gegensatz zu benutzerdefinierten Propagierern kann es deshalb sein, daß die Ausführung eines komplexen Propagierers andere ablauffähige Threads recht lange von ihrer Ausführung abhält (siehe Abschnitt 7.4). Da aber die Laufzeit eines Propagierers (bis auf Implementierungsfehler) immer endlich ist, ist Fairneß garantiert (siehe auch Abschnitt 11.6).

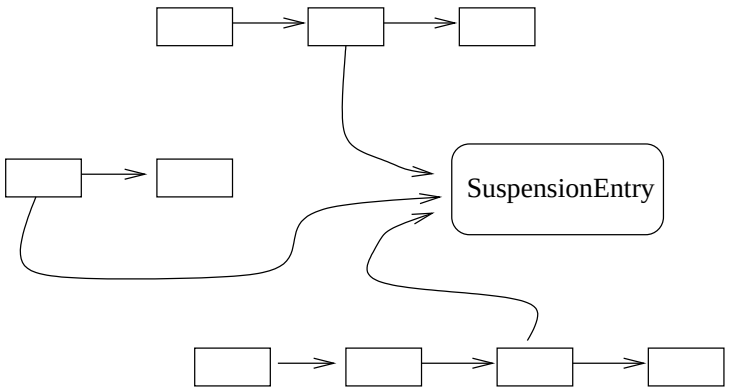
Fügt ein Propagierer für globale Variablen Basisconstraints in den Constraintspeicher hinzu (indem entsprechende Bindungen und Bereichseinschränkungen durch den Propagierer vorgenommen werden), wird auch geeignete Information im aktuellen Berechnungsraum vermerkt, um bei Deinstallation dieses Raumes den ursprünglichen Zustand wieder herzustellen. Stellt ein Propagierer fest, daß der von ihm realisierte Constraint vom aktuellen Constraintspeicher subsumiert bzw. dissubsumiert ist, wird diese Information der ausführenden abstrakten Maschine als Rückgabewert mitgeteilt. Im Gegensatz zu AKL(FD) wird in Oz der (wenn auch meist nicht sehr zeitaufwendige) Subsumtionstest bei jeder Ausführung eines Propagierers durchgeführt (in AKL(FD) muß der Programmierer sich zwischen einer propagierenden und einer nichtpropagierenden Variante entscheiden, weshalb in AKL(FD) endliche Bereiche nur partiell in das Modell lokaler Berechnungsräume integriert sind). Zusätzlich informiert der Propagierer die abtrakte Maschine, auf welche Weise die Variablen evtl. geändert wurden, damit die entsprechenden Suspensionslisten inspiziert werden können (siehe Abschnitt 11.2).

Mehrfaches Wecken eines Propagierers vor dessen Ausführung kann dadurch vermieden werden, daß es für einen Propagierer einen einzigen Suspensionseintrag gibt, auf den Verweise in den Suspensionlisten eingetragen sind (siehe Abschnitt 11.2 und Abbildung 11.5). In diesem Eintrag kann dann vermerkt werden, ob der Propagierer z. B. bereits geweckt wurde. Zusätzlich wird hierdurch der Speicherverbrauch gering gehalten (eine ähnliche Optimierung gibt es in Oz auch für Konditionale und weitere Konstrukte; siehe [Sch98a]).

```
TYPE SuspensionEntry RECORD
                        target: ↑Space;
                        args:   ↑list of FDVars;
                        arity:  ↑integer;
                        prop:   ↑procedure;
                        tag:    ↑(DEAD, SUSPENDED, WOKEN);
                     END
```

Der Eintrag `target` bezeichnet den Berechnungsraum, in dem der Propagierer auszuführen ist. Die Argumente des Propagierers sind unter `args`, seine Stelligkeit unter `arity` und seine Code-Adresse unter `prop` vermerkt. Da die Argumente eines Propagierers in einer Liste verwaltet werden, kann ein Propagierer mit einer unterschiedlichen Anzahl von Argumenten ausgeführt werden. Dadurch ist es möglich, daß ein einzelner Propagierer eine ganze Reihe von Projektoren implementiert, die sich nur in ihrer Stelligkeit unterscheiden. In DFKI Oz können Argumente beliebige Datenstrukturen sein. Für Indexicals muß hierfür immer neuer Code erzeugt werden, oder aber eine Kombination aus Bibliotheks-Indexicals verwendet werden (was aber zu weniger Propagierung führen kann; siehe Abschnitt 4.1). Suspendiert ein Propagierer, so ist das Feld

**Abbildung 11.5** Verschiedene Suspensionslisten und ein Suspensionseintrag




`tag` auf `SUSPENDED` gesetzt. Wird ein Propagierer geweckt, so wird dieses Feld auf `WOKEN` gesetzt. Die abstrakte Maschine wird beim Inspizieren der Suspensionslisten solche Einträge ignorieren, da dann der entsprechende Propagierer schon zum Ausführen in die entsprechende Warteschlange eingetragen wurde. Das Feld `tag` wird auf `DEAD` gesetzt, wenn der Propagierer Subsumtion bzw. Dissubsumtion signalisiert. In diesem Fall wird der Eintrag und die auf ihn zeigenden Verweise bei der nächsten Inspektion gelöscht.

### 11.4.1 Funktionalität in Propagierern

In diesem Abschnitt gehen wir kurz auf die bereitstehende Funktionalität ein, die zur Realisierung der Propagierungsfunktionen notwendig ist. Funktionalität für spezielle Propagierer wird in Abschnitt 11.7 vorgestellt.

Zum Zugriff auf die verschiedenen Kennwerte eines Bereichs wie sein Minimum oder auf den Bereich selbst werden eine Reihe von Funktionen zur Verfügung gestellt. Diese Funktionalität steht dem Benutzer auch über die reflektiven Prozeduren des FD-Systems zur Verfügung (wie `FD.reflect.min`; siehe Abschnitt 7.3.3). Auch die nötige Funktionalität, um Basisconstraints durch Manipulation von Bereichen zu realisieren, ist vorhanden.

Jeder Propagierer muß eine Reihe von Typüberprüfungen durchführen, bevor er mit der Ausführung des eigentlichen Propagierungsalgorithmus beginnen kann. Hierbei kann es zum einen vorkommen, daß ein Argument von einem falschen Typ ist (z. B. wird der Propagierer mit einem Atom anstatt mit einer FD-Variablen ausgeführt). In diesem Fall indiziert der Propagierer einen Laufzeitfehler. Sind zum anderen für ein Argument, das zum Beispiel eine FD-Variable sein soll, noch keine Bereichsconstraints im Constraintspeicher enthalten, so suspendiert der Propagierer.

Neben dem Typtest gibt es auch noch einen Test auf Gleichheit zwischen Variablen. Dies ist notwendig, da die meisten Propagierer Gleichheit zwischen Variablen für ihre Propagierung berücksichtigen müssen. Zum Beispiel muß ein Propagierer für `X + Y =: Z` das gleiche Propagierungsverhalten wie für `2*X =: Z` besitzen, wenn die Variablen `X` und `Y` gleichgesetzt werden. Die Propagierungsalgorithmen müssen diese Gleichheit dann bei jeder Ausführung berücksichtigen. Eine Alternative ist es, den Propagierer bei dem erstmaligen Erkennen von Gleichheit durch

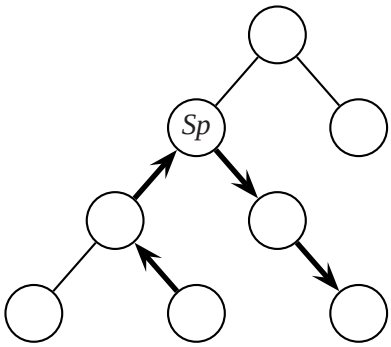
die Ausführung eines anderen zu ersetzen (siehe Abschnitt 11.7.4) oder die Propagiererargumente entsprechend zu verändern.

### 11.4.2 Aufwecken und Ausführen von Propagierern

Abhängig von der Ausführung eines Propagierers inspiziert die abstrakte Maschine die Suspensionslisten einer FD-Variablen. Die abstrakte Maschine trägt diejenigen Suspensionseinträge in die globale Warteschlange zur Propagiererausführung ein, deren Feld `tag` auf `SUSPENDED` gesetzt ist und deren Berechnungsraum (im Feld `target`) unterhalb des aktuellen liegt oder mit dem aktuellen übereinstimmt (da die Constraintspeicher in untergeordneten Berechnungsräumen die Constraints des aktuellen Speichers erben, müssen auch die dort enthaltenen Propagierer geweckt werden).

Zur Ausführung eines Propagierers wird zuerst der entsprechende Suspensionseintrag aus der globalen Warteschlange extrahiert. Stimmt der Berechnungsraum des Eintrags mit dem aktuellen überein, so kann aus dem Eintrag sofort der Aufruf des Propagierers zusammengestellt werden. Ansonsten ist der Propagierer in einem anderen Berechnungsraum auszuführen. Dazu wird der nächste gemeinsame Vorgänger *Sp* des zu deinstallierenden und des zu installierenden Berechnungsraumes in der Hierarchie bestimmt (siehe Abbildung 11.6). Dann werden alle Berechnungsräume von dem aktuellen Berechnungsraum bis zu *Sp* hin deinstalliert. Für jeden vermerkten Eintrag für einen zu deinstallierenden Berechnungsraum wird die betroffene Variable extrahiert und der aktuelle Bereich der Variablen und evtl. deren aktuelle Bindung in einer speziellen Datenstruktur des Berechnungsraumes für eine evtl. spätere Installation gespeichert. Anschließend wird gegebenenfalls die aktuelle Bindung der Variablen wieder zurückgenommen und der alte Bereich restauriert. Nach Erreichen des gemeinsamen Vorgängers *Sp*, werden von dort bis zum Ziel (in `target`) Berechnungsräume installiert. Dies geschieht mit Hilfe der bei früheren Deinstallationen angelegten Datenstrukturen. Letztendlich kann dann der Propagierer ausgeführt werden.

**Abbildung 11.6** Installation eines Berechnunsgraumes



## 11.5 Optimierungen

Die in den vorangehenden Abschnitten vorgestellte Implementierung der FD-Constraints in der abstrakten Maschine kann an verschiedenen Stellen optimiert werden. Alle diese Optimierungen sind in DFKI Oz realisiert.

### 11.5.1 Optimierungen innerhalb von Propagierern

Im bisherigen Modell erfolgen Typtests in einem Propagierer bei jeder Ausführung. Jedoch sind diese Tests nur beim ersten Ausführen eines Propagierers notwendig, da für die Argumente später nur stärkere Constraints im Constraintspeicher hinzukommen können. Deshalb wird ein Propagierer in einen Kopf- und einen Rumpfteil aufgeteilt. Beide Teile sind jeweils als C++-Prozedur implementiert. Die notwendigen Tests werden jetzt nur im Kopfteil durchgeführt. Erst wenn diese erfüllt sind, wird der Propagiererrumpf aufgerufen, der den eigentlichen Propagierungsalgorithmus enthält. Solange die Tests nicht erfüllt sind, werden Suspensionen auf den Kopf eingetragen. Sind die Tests erfüllt, werden zukünftige Suspensionen immer nur auf den Rumpf eingetragen. Im weiteren wird mit dem Begriff Propagierer immer der Rumpfteil bezeichnet.

In den vorangehenden Abschnitten arbeiteten die Propagierer immer direkt auf FD-Variablen und mit der hierfür verfügbaren Funktionalität (einschließlich Tests, ob eine globale Variable betroffen ist). Dieses allgemeine Vorgehen ist aber nicht notwendig. Stattdessen werden beim Betreten eines Propagierers neue Datenstrukturen zur Berechnung angelegt. Ist ein Argument des Propagierers eine globale Variable, so wird in diese Datenstruktur an eine passende Stelle der Bereich des Arguments kopiert. Ist ein Argument eine lokale Variable, wird lediglich ein Verweis auf den Bereich der Variablen eingetragen. Für lokale Variablen wird zusätzlich noch die initiale Größe vermerkt. Die Berechnung innerhalb des Propagierers findet dann auf diesen Datenstrukturen statt. Ist der Fixpunkt der Berechnung dieses Propagierers erreicht (wir müssen Idempotenz garantieren), müssen für globale Variablen die Bereiche evtl. reduziert werden. Nur in diesen Fällen wird eine vorgenommene Bindung oder Bereichsreduktion für die Deinstallation vermerkt. Um auf lokale Variablen suspendierende Propagierer zu wecken, wird die gespeicherte Bereichsgröße mit der aktuellen verglichen (für globale Variablen gibt es ja noch das Original, mit dem verglichen werden kann). Hat sich die Größe verändert, wird ein Wecken von Propagierern veranlaßt, da die lokale Variable durch den Propagierer stärker eingeschränkt wurde.

Bisher wurde der Test auf Variablengleichheit bei jeder Ausführung eines Propagierers durchgeführt. Jedoch muß dieser Test nur gemacht werden, wenn zuvor eine Unifikation stattgefunden hat. Wird die Ausführung einer Unifikation geeignet in der abstrakten Maschine vermerkt, können unnötige Gleichheitstests vermieden werden.

Eine weitere Optimierung betrifft eine ganze Reihe von Algorithmen zur Propagierung. Die Idee hierbei ist es, daß nur diejenigen FD-Variablen von dem Algorithmus betrachtet werden, deren Bereich sich seit der letzten Ausführung des Propagierers geändert hat (diese Idee geht auf [VDT92] zurück). Dies kann erreicht werden, indem ein Propagierer zusätzliche Argumente erhält, die die Größe der Bereiche enthalten. Zudem können die FD-Variablen, die bereits determiniert sind oder die das Propagierungsverhalten nicht mehr beeinflussen können, als Argument gelöscht werden. In DFKI Oz wird für viele Propagierer so vorgegangen.

### 11.5.2 Lokale Propagierung

Im bisher vorgestellten Modell wird auch für alle im aktuellen Berechnungsraum auszuführenden Propagierer ein Test auf Lokalität gemacht. Dabei heißt ein Propagierer *lokal* zu einem Berechnungsraum *Sp*, wenn er in *Sp* erzeugt wurde. Zudem wird der generelle Ausführungsmechanismus der abstrakten Maschine verwendet, der für Oz außer Propagierern noch beliebige Threads ausführen muß und somit für Propagierer unnötig berechnungsintensiv ist. Zudem können die meisten kombinatorischen Probleme ohne eine Hierarchie von Berechnungsräumen modelliert werden. Deshalb werden die Propagierer des aktuellen Berechnungsraumes mit einem effizienteren Mechanismus ausgeführt, der sogenannten *lokalen Propagierung*. Dazu besitzt jeder Berechnungsraum eine eigene Datenstruktur (eine Warteschlange), in der Suspensionseinträge gespeichert werden.

Beim Wecken von Propagierern werden die Suspensionseinträge für den aktuellen Berechnungsraum in diese eigene Warteschlange eingetragen. Nur die übrigen Suspensionseinträge gelangen zum Ausführen in die globale Datenstruktur der abstrakten Maschine. Nach dem Wecken der Propagierer werden die lokalen Propagierer zuerst (effizient) ausgeführt. Dabei können natürlich wieder weitere Propagierer geweckt werden etc. Erst wenn keine lokalen Propagierer mehr ausführbar sind, wird der allgemeine Mechanismus zur Ausführung von Propagierern verwendet.

Im System AKL(FD) [Car95], in dem auch lokale Berechnungsräume zu berücksichtigen sind, wird diese Optimierung nicht eingesetzt.

## 11.6 Fairneß

Für Threads wird Fairneß erreicht, indem ein Thread (sein Anweisungsstapel) von der abstrakten Maschine solange ausgeführt wird, bis er entweder terminiert oder eine vorgegebene Zeitscheibe abgelaufen ist. Da die Ausführung eines einzelnen Propagierers terminiert – was aber bei komplexen Propagierern länger dauern kann als die für Threads sonst zur Verfügung stehende Zeitscheibe – ist Fairneß bei der Integration von Propagierern gesichert.

Die Forderung nach fairer Berechnung bei Verwendung lokaler Propagierung führt dazu, daß konzeptuell die Ausführung der in der jeweiligen lokalen Warteschlange enthaltenen Propagierer selbst als Thread angesehen wird. Ist die verfügbare Zeitscheibe abgelaufen, muß der aktuelle Inhalt der lokalen Warteschlange gesichert werden, da sich der als nächstes auszuführende Thread in einem anderen Berechnungsraum befinden kann. Bei Installation eines Berechnungsraumes wird der Inhalt der lokalen Warteschlange wieder restauriert.

## 11.7 Erweiterungen

In diesem Abschnitt werden Erweiterungen der abstrakten Maschine und der Propagierer vorgestellt. Diese Erweiterungen betreffen reifizierte Constraints und die verzögerte Ausführung von Propagierern. Die zwei letzten Abschnitte zeigen, wie Propagierer von gerade ausgeführten Propagierern erzeugt werden können und wie man Propagierer von Oz aus steuern kann.

### 11.7.1 Reifizierte Constraints

Reifizierte Constraints (siehe Abschnitt 4.4) werden von der abstrakten Maschine direkt unterstützt. Dazu steht ein generischer Mechanismus zur Verfügung, um beliebige Constraints zu refizieren, wenn ihre Propagierer in der Implementierung vorhanden sind. Sei der Constraint $c$ in die 0/1-wertige Variable $x$ reifiziert, also $c \leftrightarrow x = 1$. Wird der Propagierer $p$, der den Projektor für $c \leftrightarrow x = 1$ implementiert, ausgeführt, wird zuerst getestet, ob $x$ an Eins gebunden ist. In diesem Fall wird der Propagierer $p$ durch einen Propagierer $p_c$ für $c$ ersetzt (siehe Abschnitt 11.7.4). Als Ergebniswert von $p$ wird das Ergebnis des Propagierers $p_c$ verwendet. Entsprechend wird $p$ durch einen Propagierer für $\neg c$ ersetzt, wenn $x$ an Null gebunden ist.

Ist $x$ nicht gebunden, wird durch den Propagierer $p$ getestet ob $c$ subsumiert oder dissubsumiert ist. In Fällen wie $y \leq z$ sind zwei einfache Tests ausreichend (siehe Abschnitt 4.1). Diese Tests können effizient implementiert werden. Ist einer dieser Tests erfolgreich, wird die 0/1-wertige Variable entsprechend gebunden. In allen vier Fällen kann $p$ gelöscht werden.

Für andere Constraints sind diese Tests nicht mehr so einfach. Gilt zum Beispiel `Y` $\in [1\#5\ 9\#10]$, so muß für `B = (Y=:7)` die Variable `B` an Eins gebunden werden. Dies würde man durch einen einfachen Test der oberen und unteren Schranke des Bereichs von `Y` nicht erreichen, da sowohl `Y` $< 7$ als auch `Y` $> 7$ weder subsumiert noch dissubsumiert sind.

In diesen Fällen wird der Propagierer $p_c$ spekulativ ausgeführt. Hierzu werden von den Bereichen der Argumente des Propagierers Kopien angelegt. Der Propagierungsalgorithmus rechnet dann mit diesen Kopien anstatt mit den originalen Argumenten. Wird ein leerer Bereich erzeugt, so wird nicht `failed` zurückgegeben, sondern die 0/1-wertige Variable $x$ an Null gebunden. Wird Subsumtion erkannt, wird $x$ an Eins gebunden. In beiden Fällen ist der Propagierer $p$ subsumiert und gibt `entailed` als Ergebnis zurück.

Diese spekulative Berechnung erlaubt die Reifikation beliebiger durch Propagierer realisierter Constraints. In Systemen wie `clp(FD)` oder AKL(FD) ist eine solche Implementierung durch spekulative Berechnung prinzipiell nicht möglich, da dies mit dem Konzept der verwendeten Indexicals nicht vereinbar ist.

### 11.7.2 Prioritäten

Oftmals unterscheiden sich verschiedene Propagierer sehr stark in ihrer Ausführungszeit. Dies kann von der Anzahl der Argumente oder von der Stärke des Propagierungsalgorithmus abhängen. Zum Beispiel wird die Ausführung einer reduktionsvollständigen Variante von `FD.distinct` länger dauern als die von `X <: Y`. Deshalb ist es möglich, Propagierer mit unterschiedlichen Prioritäten zu versehen. Entsprechend den Prioritäten für beliebige Threads in DFKI Oz (siehe [MMP+97]), kann man Propagierer mit einer Priorität *low*, *medium* oder *high* versehen und es gibt drei Warteschlangen. Die Priorität *medium* ist der Regelfall für Propagierer. Nur lokale Propagierer mit der Priorität *medium* werden für die lokale Propagierung herangezogen. In der Regel sind dies die meisten Propagierer, die während einer Lösungssuche erzeugt werden. Besitzt ein Propagierer eine andere Priorität oder ist er nicht lokal, wird er in der seiner Priorität entsprechenden Warteschlange zur Wiederausführung eingetragen. In DFKI Oz spielen unterschiedliche Prioritäten von Propagierern jedoch noch keine Rolle.

### 11.7.3 Nicht-monotone Propagierungsfunktionen

In Kapitel 5 wurden nicht-monotone Propagierungsfunktionen vorgestellt. Um die entsprechenden Algorithmen trotzdem für die Implementierung von Propagierern verwenden zu können, muß von der abstrakten Maschine sichergestellt werden, daß der Fixpunkt der Propagierung aller in einem Berechnungsraum enthaltenen Propagierer immer eindeutig ist, also *Konfluenz* gilt (modulo logischer Äquivalenz).

Hierzu erhält jeder Propagierer, dessen Kopfteil zum ersten Mal ausgeführt wird, eine eindeutige, von der abstrakten Maschine vergebene Nummer. Suspensionseinträge für Propagierer werden durch diese Nummer und eine Marke erweitert, die den Propagierer als nicht-monoton kennzeichnet. Soll ein solcher Propagierer wieder ausgeführt werden, wird ein Eintrag in eine entsprechende Datenstruktur *d* eingetragen, die pro Berechnungsraum einmal vorhanden ist. In dieser Datenstruktur sind die Einträge nach ihrer eindeutigen Nummer geordnet. Sind keine anderen Propagierer mehr ausführbar, so werden die zu den Einträgen in *d* korrespondierenden Propagierer, ihrer Numerierung folgend, ausgeführt. Dies kann zur Folge haben, daß erneut Weckaufträge in entsprechende Datenstrukturen eingetragen werden und die zweistufige Ausführung beginnt von neuem. Sind sonst keine weiteren nicht-monotonen Konstrukte vorhanden, ist damit ist die gewünschte Konfluenz garantiert.

### 11.7.4 Propagierer, die Propagierer erzeugen

In manchen Fällen soll der gerade ausgeführte Propagierer durch einen anderen ersetzt werden. Dies ist zum Beispiel für reifizierte Constraints der Fall, deren kontrollierende 0/1-wertige Variable determiniert wurde. Hierzu stellt die abstrakte Maschine entsprechende Funktionalität bereit. Der erzeugte Propagierer wird in die (lokale) Warteschlange zur Wiederausführung eingetragen, so daß der erzeugte Propagierer erst nach der Beendigung des laufenden Propagierers ausgeführt wird (hier trifft also Abbildung 11.1 nicht zu). Der ersetzende Propagierer bestimmt den Ergebniswert des ersetzten Propagierers. Diese Technik wird auch dazu verwendet, um Propagierer durch effizientere Spezialvarianten zu ersetzen (zum Beispiel kann der Propagierer für `1*X <: Y` durch einen Spezialpropagierer für `X<:Y` ersetzt werden).

Diese Technik wird zum Beispiel auch für aufgabenorientierte Serialisierer verwendet (siehe Abschnitt 6.3), um notwendigerweise geltende Ordnungen zu fixieren (dabei sind Serialiserer zwar keine Propagierer, können aber durch die gleiche Implementierungstechnik realisiert werden; siehe Abschnitt 11.7.5).

### 11.7.5 Programmgesteuerte Propagierer

Propagierer können nicht nur auf FD-Variablen suspendieren, sondern auch auf ungebundene Variablen. Sobald eine solche Variable gebunden wird, wird der Propagierer geweckt. Suspendiert ein Propagierer auf einen Strom (eine Liste mit offenem Ende), so kann man auf diese Art von Oz aus mit einem Propagierer kommunizieren und der Propagierer kann von einem Oz-Programm gesteuert werden. Solche Propagierer heißen *programmgesteuerte Propagierer*.

Als ein Beispiel betrachten wir einen Propagierer *p*, der auf den Strom `Xs` suspendiert. Wird

nun `Xs` an `message(M Out)|Xr` gebunden, kann *p* für seinen Propagierungsalgorithmus Informationen verwenden, die über die Variable `M` zugänglich sind. An die Variable `Out` kann *p* dann Information binden, die von einem Oz-Programm genutzt werden kann. Der Propagierer *p* suspendiert dann auf die Variable `Xr` und ist damit für eine nächste Kommunikation bereit. Auf diese Weise kann zum Beispiel ein `FD.distinct` realisiert werden, das dynamisch mehr Argumente erhalten kann (zur Erinnerung: `FD.distinct` besagt, daß die Werte einer Menge von Variablen paarweise verschieden sein müssen). Damit wird die Erzeugung zusätzlicher (teilweise redundanter) Propagierer vermieden.

Eine weitere Anwendung dieser Technik sind die in Kapitel 6 vorgestellten Serialisierer, für die in DFKI Oz die Implementierungstechnik von Propagierern verwendet wird. Ein Serialisierer besteht damit aus einem Oz-Programm und einer C++-Prozedur, die über einen Strom miteinander kommunizieren können. Das Oz-Programm kommuniziert der C++-Prozedur auf dem Strom die zuletzt in einem Wahlpunkt getroffene Ordnungsentscheidung in einer geeigneten Datenstruktur. Diese Information wird dazu verwendet, schon getroffene Ordnungsentscheidungen zu vermerken (Serialisierer sind ja zustandsabhängige Distribuierungsstrategien). Umgekehrt stellt die C++-Prozedur den Vorschlag für den nächsten Distribuierungsschritt dem Oz-Programm auf dem Strom zur Verfügung.

Als weitere Anwendungen von Strömen kann man sich die direkte Kontrolle des Propagierungsverhaltens eines Propagierers vorstellen. Außerdem kann der Propagierer, zum Beispiel für Debugging-Zwecke, Informationen über seinen aktuellen Berechnungszustand nach außen verfügbar machen.

## 11.8 Ausblick

Ein Nachteil der bisher vorgestellten Implementierung des FD-Systems ist es, daß zur Entwicklung neuer Propagierer der Programmierer Zugriff auf die Funktionalität der abstrakten Maschine haben muß. Dies kann durch eine geeignete Schnittstelle zu der abstrakten Maschine geändert werden. Diese Schnittstelle stellt passende Abstraktionen zur Verfügung, um die in den vorangehenden Abschnitten angeführte Funktionalität zu implementieren. Durch diese Entkopplung von der abstrakten Maschine kann der Code für Propagierer dynamisch gebunden werden. Diese Idee führte zur Entwicklung des sogenannten *Constraint Propagator Interface* [MW97a, MW96, MW97b]. Auch sind dort Propagierer nicht mehr als C++-Prozeduren sondern als C++-Objekte implementiert. Dadurch wird es zum Beispiel auch einfach, Propagierer mit einem Zustand zu versehen.

Die in diesem Kapitel vorgestellte Implementierungstechnik läßt sich auch für andere variablenzentrierte Constraintsysteme verwenden. Dies wurde in [Mül95] für Intervall-Constraints für reelle Zahlen gezeigt. Eine Erweiterung des Constraint Propagator Interface ermöglicht das Rechnen mit Mengen-Constraints über natürlichen Zahlen [MM97].

# Kapitel 12

# Konstruktive Disjunktion

In den vorangehenden Kapiteln haben wir gesehen, wie häufig disjunktive Constraints in der Praxis vorkommen. Diese wurden durch Propagierer für reifizierte Constraints oder Spezialpropagierer realisiert. Während man bei Verwendung reifizierter Constraints rasch zu einer lauffähigen Realisierung kommt, wird doch der Suchraum erst dann eingeschränkt, wenn alle Klauseln einer Disjunktion bis auf eine dissubsumiert sind. *Konstruktive Disjunktion* [VSD91, VSD93] kann den Suchraum bereits früher einschränken. Die prinzipielle Idee hierbei ist es, gemeinsame Information aus den Klauseln zu extrahieren und zur Reduktion des Suchraumes zu verwenden. Für den allgemeinen Fall kann dies nicht mehr durch einen einzelnen Propagierer geleistet werden, sondern es muß ein neues Kontrollkonstrukt in Oz integriert werden.

Sei $c_1 \vee \ldots \vee c_n$ der mit konstruktiver Disjunktion zu realisierende Constraint und sei $c$ der Constraintspeicher, des die konstruktive Disjunktion enthaltenden Berechnungsraumes *Sp*. Konzeptuell werden alle Propagierer, die einen Constraint $c_i$ realisieren, jeweils in einem Berechnungsraum erzeugt, der direkt dem Berechnungsraum *Sp* untergeordnet ist. Gilt für ein $c_i$, daß alle $c_i$ realisierenden Propagierer subsumiert sind, so reduziert die konstruktive Disjunktion zur leeren Anweisung. Gilt für $n-1$ Constraints unter $c_1$ bis $c_n$, daß der jeweils erzeugte Berechnungsraum fehlgeschlagen ist, wird der verbleibende Berechnungsraum mit dem Berechnungsraum *Sp* verschmolzen (die Basisconstraints und Propagierer des untergeordneten Berechnungsraumes werden zum übergeordneten Berechnungsraum *Sp* hinzugefügt). Ansonsten wird der Suchraum wie folgt eingeschränkt. Gilt $c \wedge c_1 \models b, \ldots, c \wedge c_n \models b$ für einen Basisconstraint $b$, so wird der von allen $c \wedge c_i$ subsumierte Constraint $b$ zu dem Constraintspeicher $c$ hinzugefügt.

Prinzipiell ist konstruktive Disjunktion also unabhängig von einer konkreten Constraintstruktur definiert. In Oz wird sie aber nur für Constraints über den ganzen Zahlen unterstützt, wobei die konkrete Syntax wie folgt definiert ist.

**`condis`** $C_1$ `[]` $C_2$ `...` `[]` $C_n$ **`end`**

Hierbei gilt $n \geq 1$ und die Klauseln $C_i$ dürfen nur Infixanweisungen für Constraints enthalten (siehe Abschnitt 7.3.2).

Wir betrachten jetzt das folgende Beispiel aus dem Bereich des Scheduling. Es seien zwei Aufgaben mit den Startzeitpunkten $X$ und $Y$ und der Ausführungszeit 8 gegeben. Der disjunktive

Constraint

$$X + 8 \leq Y \ \vee \ Y + 8 \leq X$$

besagt, daß die beiden Aufgaben sich zeitlich nicht überlappen dürfen (da sie zum Beispiel die gleiche Ressource verwenden). Dieses Beispiel läßt sich durch

```
condis X+8=<:Y  [] Y+8=<:X end
```

realisieren. Gilt $\mathtt{X}, \mathtt{Y} \in 0\#10$, so kann durch Realisierung mit Propagierern für reifizierte Constraints nicht mehr Information gewonnen werden. Da jedoch für die gegebenen Bereichsconstraints aus der linken Klausel $\mathtt{X} \in 0\#2$ und aus der rechten $\mathtt{X} \in 8\#10$ folgt, kann der Constraintspeicher durch $\mathtt{X} \in [0\#2\ 8\#10]$ verstärkt werden (und genauso für Y).

Wegen der starken Propagierung hat konstruktive Disjunktion in der Regel eine größere Laufzeit als zum Beispiel entsprechende reifizierte Constraints. Jedoch kann die vordefinierte Form von konstruktiver Disjunktion zum schnellen Programmieren eines Prototyps verwendet werden. Zeigt sich, daß ihre Verwendung von Vorteil ist, kann für diese konkrete Anwendung von konstruktiver Disjunktion ein effizienterer Spezialpropagierer realisiert werden. Es ist jedoch für ein zu lösendes Problem immer zu prüfen, ob die zusätzliche Information überhaupt nützlich ist (siehe auch [WM96]). Würden zum Beispiel durch konstruktive Disjunktion vor allem Werte innerhalb der Bereiche entfernt, so ist es nicht sinnvoll, konstruktive Disjunktion zu verwenden, wenn ansonsten nur die Grenzen von Bereichen zur Propagierung ausgenutzt werden (wie es für Scheduling der Fall ist).

## 12.1 Fallstudie

In diesem Abschnitt nehmen wir die Fallstudie zur Erstellung von Stundenplänen aus Kapitel 8 wieder auf. Wir haben dort eine Reihe von disjunktiven Constraints durch Propagierer für reifizierte Constraints realisiert. Einige konnten auch durch Propagierer für globale Constraints realisiert werden. Für C9, C12 und C14 war dies jedoch nicht möglich. Diese Constraints sind jedoch gerade von der Bauart des obigen Beispiels, so daß sie direkt durch konstruktive Disjunktion realisiert werden können.

Constraint C13 kann realisiert werden, indem eine 0/1-wertige Variable verwendet wird, die darüber Auskunft gibt, ob zwei Kurse `C1` und `C2` sich nicht überschneiden (Seite 131). Mit konstruktiver Disjunktion läßt sich dies wie folgt realisieren (`C.start` sei die Startzeit und `C.dur` die Dauer eines Kurses `C`).

```
condis B=:1
   C1.start + C1.dur >: C2.start
   C2.start + C2.dur >: C1.start
[] B=:0   C1.start + C1.dur =<: C2.start
[] B=:0   C2.start + C2.dur =<: C1.start
end
```

Verwendet man diese Formulierung mit konstruktiver Disjunktion zusammen mit globalen Constraints, kann man wieder die in Abschnitt 8.7 beschriebenenen Distribuierungsstrategien evaluieren (siehe Tabelle 12.1). Dazu seien $D_7$, $D_8$ und $D_9$ entsprechend wie in Abschnitt 8.7 defi-

niert. Durch Verwendung konstruktiver Disjunktion werden die Suchbäume mit $D_7$ und $D_8$ im

**Tabelle 12.1** Evaluierung der verschiedenen Lösungsstrategien

| Strategie | *first* | *good* | *Mon-first* | *After-first* | *Mon-good* | *After-good* |
|---|---|---|---|---|---|---|
| $D_7$ | 80 (23) | 153 (89) | 7 | 26 | 5 | 23 |
| $D_8$ | 80 (23) | 156 (94) | 7 | 26 | 5 | 19 |
| $D_9$ | 80 (23) | 144 (83) | 3 | 25 | 2 | 20 |

Vergleich zu $D_2$ und $D_4$ leicht größer. Die Ergebnisse nach der Optimierung sind jedoch besser. Insgesamt schneidet $D_9$ am besten ab, wobei gegenüber $D_6$ eine drastische Reduzierung der Suchbaumgröße eintritt. Da Strategie $D_9$ bei der Distribuierung oft mit Werten aus dem Inneren von Bereichen beginnt, ist Propagierung besonders wichtig, die Lücken in Bereichen erzeugt (wie es konstruktive Disjunktion leistet). Die Laufzeit bei Verwendung von $D_9$ ist etwa 20% größer als bei Verwendung von $D_2$.

## 12.2 Implementierung

Hier wird nur die prinzipielle Implementierungsidee in der abstrakten Maschine (siehe Kapitel 11) erklärt. Eine ausführliche Darstellung findet sich in [MW95, WM96].

Bei der Implementierung ist darauf zu achten, daß die Berechnung in den Klauseln spekulativ erfolgt. Das bedeutet, daß solange mindestens in zwei Klauseln noch kein Constraint dissubsumiert ist, die Bereiche von Variablen nur durch die Hinzunahme allen Klauseln gemeinsamer Information verändert werden dürfen. Sonst dürfen keine Bereiche von Variablen verändert werden. Zusätzlich muß ein Weg vorgesehen werden, wie eben diese gemeinsame Information herausgefunden werden kann.

Der Oz-Compiler erzeugt für Infixanweisungen für Bereichsconstraints oder Propagierer zwischen den Schlüsselworten **`condis`** und **`end`** andere Prozeduraufrufe als üblich (siehe Abschnitt 7.3.2). Zusätzlich wird für jede in einer Klausel vorkommende Variable eine neue ungebundene Variable deklariert, deren Sichtbarkeitsbereich auf die Propagierer der konstruktiven Disjunktion beschränkt ist (durch entsprechend vom Compiler erzeugte Anweisungen **`local`** `...` **`end`**). Dies ist notwendig, damit in den Klauseln nur spekulativ gerechnet wird. Zusätzlich erhält jeder Propagierer noch ein weiteres Argument, nämlich eine 0/1-wertige Variable, die pro Klausel neu deklariert wird. Diese Variable dient der Kontrolle der in der Klausel vorkommenden Propagierer. Ist sie an Null gebunden, wird der Propagierer gelöscht, indem er `entailed` als Ergebnis zurückliefert. Ist die Variable an Eins gebunden, wird der aktuelle Propagierer durch den Propagierer passend zur Infixanweisung ersetzt (siehe Abschnitt 11.7.4) – dies realisiert das sogenannte *unit-commit*. Ist ein Propagierer subsumiert, so bindet er die 0/1-wertige Variable an Eins. Ist ein Propagierer dissubsumiert, so bindet er die Variable an Null.

Das Propagierungsverhalten der konstruktiven Disjunktion wird durch einen weiteren Propagierer sichergestellt. Dieser suspendiert auf die original in den Klauseln vorkommenden FD-Variablen und auf die neu eingeführten Variablen (inkl. der 0/1-wertigen Variablen). Ändert sich eine Originalvariable, so werden die Bereiche der neuen Variablen entsprechend reduziert (dies gilt auch für Gleichheit von Variablen). Umgekehrt bildet dieser Propagierer die Vereinigung der

Bereiche von neuen Variablen in den Klauseln, die für die gleiche Originalvariable eingeführt wurden. Um diesen relativ zeitaufwendigen Prozeß effizient zu implementieren, wird bei konstruktiver Disjunktion die *lokale Propagierung* benutzt (siehe Abschnitt 11.5.2). Erst wenn die Propagierer der Klauseln ihren Fixpunkt der Propagierung erreicht haben, wird die gemeinsame Information berechnet. Über die 0/1-wertigen Variablen hat der kontrollierende Propagierer Informationen, in wieviel Klauseln schon Constraints dissubsumiert sind. Im Falle von unit-commit werden die neuen Variablen der entsprechenden Klausel mit den entsprechenden Originalvariablen unifiziert.

Über einen allgemeinen Mechanismus können von beliebigen Propagierern Versionen erstellt werden, die für konstruktive Disjunktion verwendet werden können. Durch deren direkte Verwendung (also ohne das syntaktische Konstrukt **`condis ... end`**) in Kombination mit dem eine konstruktive Disjunktion kontrollierenden Propagierer können beliebige disjunktive Constraints konstruktiv realisiert werden (siehe [MW95, WM96]).

## 12.3 Historische Anmerkungen

Konstruktive Disjunktion wurde mit [VSD91] und [VSD93] in die Constraintprogrammierung eingeführt. Der Unterschied zur konstruktiven Disjunktion in Oz besteht darin, daß in [VSD93] die gemeinsame Information aus $c(P) \wedge c \wedge c_1$ bis $c(P) \wedge c \wedge c_n$ extrahiert wird (also auch die in *Sp* enthaltenen Propagierer *P* berücksichtigt werden, wobei *Sp* der die konstruktive Disjunktion enthaltende Berechnungsraum ist; $c(P)$ bezeichnet den Constraint, der durch die Propagierermenge *P* implementiert wird). Im gleichen Jahr wurden auch die Implementierungen von [lPW93] und [JS93] vorgestellt. In [lPW93] wird eine allgemeine Erweiterung des ECL$^i$PS$^e$-Systems [ECR96] geschildert, die auch zur Realisierung von konstruktiver Disjunktion geeignet ist. In [CC95] wurde konstruktive Disjunktion schließlich in das sogenannte Indexical-Schema [VSD91] integriert. Es wird die starke Variante aus [VSD93] wie auch eine schwächere wie in Oz realisiert. Letztere berücksichtigt aber keine Gleichheitsconstraints, und das Implementierungsschema in Oz ist allgemeiner, da durch geringe Änderungen jeder beliebige Propagierer für konstruktive Disjunktion verwendet werden kann (siehe Abschnitt 12.2).

# Kapitel 13

# Verwandte Systeme und Evaluierung

In diesem Kapitel werden eine Reihe von FD-Systemen vorgestellt. Das Hauptziel des Kapitels ist es aber, zu zeigen, daß Oz sowohl expressiv genug ist, um eine Reihe von kombinatorischen Problemen zu modellieren, als auch effizient genug ist, sie zu lösen. Wir können zeigen, daß Oz kompetitiv zu den besten verfügbaren Constraintsystemen ist. Dazu verwenden wir einige Standardprobleme und verschiedene Schedulingprobleme. Insbesondere der Vergleich zur Lösung von Schedulingproblemen und die Kapitel 8, 9 und 10 belegen die Eignung von Oz zur Lösung von Anwendungsproblemen aus der Praxis.

In Abschnitt 13.1 werden Systeme vorgestellt, die einen großen Einfluß auf die Constraintprogrammierung mit Constraints über den ganzen Zahlen (finite domain constraints) und/oder die Entwicklung des FD-Systems von Oz hatten. Ein Vergleich mit Oz findet sich an entsprechenden Stellen auch oft in vorangehenden Kapiteln. Wir verzichten auf einen Vergleich mit `cc(FD)` [VSD95], da eine Implementierung nie verfügbar gewesen ist. AKL(FD) [CCJ95]ist wegen seiner Einbettung in die nebenläufige Constraintsprache AKL [HJ90, Jan94] aufgenommen, in der ähnliche Implementierungsaspekte wie in Oz zu berücksichtigen waren. Abschnitt 13.2 enthält einen Vergleich der Systeme bzgl. ihrer Laufzeit für eine Menge von kleinen Beispielproblemen. Abschnitt 13.3 enthält einen ausführlichen Vergleich von ILOG SCHEDULER [ILOG96a] und CLAIRE [CL94b] mit Oz für eine Reihe von Schedulingproblemen.

## 13.1 Verwandte Systeme

### 13.1.1 CHIP

CHIP war das erste CLP-System (*constraint logic pogramming*), das das Lösen von kombinatorischen Problemen über den ganzen Zahlen unterstützte [DVS+88, COS96]. Neben solchen Problemen können auch Probleme über den rationalen Zahlen und booleschen Werten gelöst werden.

Wie in Oz sind nur nicht-negative ganze Zahlen für Bereiche von Variablen erlaubt. Arithmetische Constraints wie Gleichungen und Ungleichungen sind durch spezielle Notation verfügbar. Zusätzlich zu den arithmetischen Constraints wurden in CHIP erstmals *symbolische Constraints*

wie *element* oder *alldifferent* zur Modellierung von speziellen Anwendungen eingeführt.

Reifikation (siehe Abschnitt 4.4) ist in CHIP nur eingeschränkt über *Konditionale* verfügbar. Im Bedingungsteil eines Konditionals kann nur eine Gleichung bzw. Ungleichung mit zwei FD-Variablen und einer Konstanten vorkommen. Für diese Constraints wird auf Subsumtion bzw. Dissubsumtion getestet.

Der Benutzer kann eigene Constraints über sogenannte *Dämonen* (engl. *demons*) definieren. Hierzu wird an Variablen vermerkt, daß bei bestimmten Änderungen der Variablenbereiche (wie zum Beispiel einer Verkleinerung der oberen Bereichsgrenze) entsprechende CHIP-Klauseln aufgerufen werden sollen. Zur Implementierung wird das sogenannte *coroutining* verwendet (siehe zum Beispiel [Car87] oder [Hol90]).

Der Schwerpunkt von CHIP liegt auf einer Reihe von globalen Constraints (siehe Abschnitt 4.2), die über Parameter an anwendungsspezifische Erfordernisse angepaßt werden können (siehe auch [DSV90, AB93, BC94]). Anwendungsgebiete der globalen Constraints sind zum Beispiel disjunktive oder kumulative Schedulingprobleme (*cumulative*), mehrdimensionale Plazierungsprobleme (*diffn*) oder Transportprobleme (*cycle*). Durch die Integration von komplexen Algorithmen aus der Graphentheorie oder dem Operations Research konnte gezeigt werden, daß Constraintprogrammierung in der Effizienz selbst mit Spezialverfahren konkurrieren kann. Über die Algorithmen, die zur Implementierung der globalen Constraints in CHIP verwendet werden, ist jedoch praktisch nichts veröffentlicht.

### 13.1.2 ECL$^i$PS$^e$

ECL$^i$PS$^e$ [ECR96] ging (über SEPIA [MAC$^+$89] als Zwischenstation) aus CHIP hervor und enthält neue Implementierungstechniken. Aufbauend auf der Syntax von CHIP, ermöglicht es ECL$^i$PS$^e$, in Bereichen auch negative ganze Zahlen zu verwenden. Neben Problemen über den ganzen Zahlen können auch Probleme über rationalen Zahlen und Mengenconstraints gelöst werden. Sogenannte *generalised propagation* stellt einen Mechanismus ähnlich konstruktiver Disjunktion (siehe Kapitel 12) zur Verfügung (nur expressiver und damit auch meist teurer; siehe [MW95]). *Constraint Handling Rules* erlauben die einfache (aber meist auch nicht sehr effiziente) Integration neuer Constraintlösetechniken durch eine Art Termersetzungssystem [Frü95].

Es stehen einige symbolische Constraints wie *element* zur Verfügung. Seit Ende 1996 sind reifizierte Constraints für Gleichungen, Ungleichungen und boolesche Operationen verfügbar.

Neue Constraints können mit Hilfe von attributierten Variablen [Hol90] definiert werden. Hierzu muß festgelegt werden, wann der so definierte Constraint wieder ausgeführt werden soll (zum Beispiel, wenn ein Element aus einem Bereich entfernt wird). Der Benutzer muß explizit die Suspensionslisten verwalten und Constraints selbst aufwecken bzw. den ausgeführten Constraint wieder suspendieren (siehe [MW97b] für eine Erweiterung von Oz, die dies umgeht).

ECL$^i$PS$^e$ ist ein bewährtes System, das auch zur Lösung von einigen Problemen aus der industriellen Praxis eingesetzt wurde [CF95]. Die Definition neuer Constraints ist nur bedingt effizient möglich. Dies ist in etwa mit den Möglichkeiten in Oz vergleichbar, neue Constraints durch Konditionale, die verfügbaren Basisconstraints und entsprechende Funktionalität des Verbunds `FD.watch` auszudrücken (wobei in Oz das explizite Suspendieren und Aufwecken von

Constraints entfällt; siehe auch Abschnitt 7.3.3). Globale Constraints werden (außer einfachen symbolischen Constraints) nicht angeboten. Eine Weiterentwicklung von ECL$^i$PS$^e$ ist das am IC Parc in London entwickelte System ECL$^i$PS$^e$ II [WNS97], das zum Beispiel Schnittstellen für lineares Programmieren [RWH96] enthält.

### 13.1.3 `clp(FD)`

Ausgehend von einem Entwurf in [VSD91] ist `clp(FD)` die erste Implementierung eines FD-Systems, in der ein Constraint durch (mehrere) sogenannte *Indexicals* realisiert wird [DC93, CD96].

Ein Indexical ist eine reaktive funktionale Regel "*x* in *r*", wobei *x* eine FD-Variable und *r* ein Ausdruck ist, der definiert, welche Basisconstraints für *x* (abhängig vom aktuellen Constraintspeicher) vom Indexical zur Propagierung verwendet werden und zu welchem Zeitpunkt (zum Beispiel bei der Änderung einer unteren Bereichsgrenze) der Indexical wieder ausgeführt werden soll.

Die Syntax von `clp(FD)` ist an diejenige von CHIP angelehnt. Ein `clp(FD)`-Programm wird in eine Folge von abstrakten Maschinenbefehlen übersetzt und aus diesen wiederum ein ablauffähiges C-Programm gewonnen [CD95]. Durch verschiedene Optimierungen ist `clp(FD)` für einige Standardprobleme sehr effizient. Arithmetische Constraints werden in Aufrufe von Bibliotheksprozeduren übersetzt, die wiederum durch Indexicals definiert sind.

Einige symbolische Constraints wie *element* sind vorhanden. Boolesche Constraints werden durch Constraints über ganze Zahlen und 0/1-wertige Variablen modelliert [CD93]. Reifizierte Constraints sind experimentell durch zusätzlichen C-Code für einige Beispiele realisiert. In Bereichen können nur nicht-negative ganze Zahlen vorkommen.

`clp(FD)` zeigt eindrucksvoll, daß ein sogenannter *glass-box* Ansatz mit benutzerdefinierten Constraints sehr effizient sein kann. Das größte Problem für `clp(FD)` ist jedoch, daß komplexe globale Constraints nicht realisiert werden können [PL95, MW97b]. Durch die Aufspaltung eines Constraints in mehrere Indexicals geht die Möglichkeit verloren, Algorithmen über alle Argumente eines Constraints zu formulieren.

### 13.1.4 AKL(FD)

In AKL(FD) [CCJ95, Car95] wurden Indexicals in die nebenläufige Constraintsprache AKL [HJ90, Jan94] integriert. In AKL werden Berechnungsräume in sogenannten *deep guards* verwendet (in Oz entspricht dies Wächtern von Konditionalen, die durch Berechnungsräume realisiert sind; siehe Abschnitt 7.1.4).

In AKL(FD) wird das Indexical-Schema von `clp(FD)` übernommen und den Ideen von `cc(FD)` [VSD93, VSD95] folgend erweitert. So können reifizierte Constraints durch einen Test auf Subsumtion von Indexicals implementiert werden [CCD94]. Außerdem ist konstruktive Disjunktion in verschiedenen Ausprägungen vorhanden (sogenannte schwache und starke konstruktive Disjunktion; siehe auch Kapitel 12). Einige vordefinierte symbolische Constraints sind verfügbar. Durch die Möglichkeit des Subsumtionstests ist auch der *cardinality*-Kombinator

von `cc(FD)` realisierbar [VD91b, VSD95].

Während AKL (wie Oz) Suche durch Kopieren von Berechnungsräumen realisiert, ist für AKL(FD) eine spezielle experimentelle Implementierung nur für Indexicals entwickelt worden, die auf dem sogenannten Trailing basiert [Car95].

Die Implementierung der FD-Variablen und des Suspensionsmechanismus in Oz haben einige Gemeinsamkeiten mit der Implementierung von AKL(FD). Die Integration von Constraints in AKL(FD) ist jedoch nicht so vollständig wie in Oz, da in AKL(FD) ein Indexical nicht gleichzeitig zum Propagieren und Testen auf Subsumtion verwendet werden kann. Wie auch in `clp(FD)`, ist es in AKL(FD) nicht möglich, komplexere globale Constraints zu definieren. Für AKL(FD) steht nur eine experimentelle Version zur Verfügung, da das System nicht freigegeben wurde.

### 13.1.5 SICStus

[COC97] beschreibt ein FD-System, das in SICStus-Prolog [Int95] integriert wurde. Das System baut auf der in [Car95] beschriebenen Implementierung von Indexicals auf. Zusätzlich können globale Constraints über eine Prolog-basierte Schnittstelle zum System hinzugefügt werden. Reifizierte Constraints sind teilweise über geschachtelte Notation verfügbar, können aber auch über spezielle Prädikatskonstruktionen definiert werden. Im Gegensatz zu Oz können auch negative ganze Zahlen für Bereiche verwendet werden. Dem Benutzer stehen eine Reihe von Distribuierungsstrategien und einige symbolische Constraints zur Verfügung.

Arithmetische Constraints und globale Constraints stehen über entsprechende Bibliotheksaufrufe zur Verfügung und sind aus Effizienzgründen in C implementiert. Indexicals wie auch globale Constraints werden über sogenannte attributierte Variablen [Hol90] in das zugrundeliegende Prolog-System integriert.

### 13.1.6 ILOG SOLVER

ILOG SOLVER ist eine kommerzielle C++-Bibliothek, in der die Konzepte aus der Constraintprogrammierung integriert wurden [Pug94, ILOG96b]. Neben Constraints über den ganzen Zahlen können auch Constraints über Intervallen von Fließkommazahlen und endliche Mengen gelöst werden. Die Bereiche von FD-Variablen können auch negative ganze Zahlen enthalten.

ILOG SOLVER bietet eine umfangreiche Funktionalität zur Lösung von kombinatorischen Problemen wie reifizierte Constraints (hier *Metaconstraints* genannt), verschiedene symbolische Constraints, unterschiedliche Distribuierungsstrategien und Suchtechniken an. Über die zusätzliche Bibliothek ILOG SCHEDULER [ILOG96a] sind globale Constraints und Distribuierungsstrategien für Scheduling verfügbar (siehe auch Abschnitt 9.4).

Constraintprogramme bestehen aus C++-Code, wobei sowohl FD-Variablen als auch Propagierer als Objekte mit entsprechenden Methoden definiert sind. Der Benutzer kann eigene Propagierer in der Implementierungssprache C++ selbst definieren.

ILOG SOLVER zeigt, daß Constraintprogrammierung auch in konventionellere Programmiersprachen eingebettet werden kann.

## 13.2 Effizienzvergleich der Systeme

In diesem Abschnitt werden die in Abschnitt 13.1 vorgestellten Systeme anhand einiger Beispielprobleme verglichen. Dies sind zum einen Standardprobleme, wie sie in Vergleichen häufig verwendet werden, und zum anderen spezielle Probleme, um die Effizienz von Propagierung zu testen.

Zu Beginn einige prinzipielle Worte zu einem solchen Vergleich. Die Codierung in Oz oder anderen Systemen enthält oft globale oder symbolische Constraints, die nicht in allen verglichenen Systemen vorhanden sind. Dieses Vorgehen ist jedoch gerechtfertigt, da globale Constraints gerade zur Effizienzsteigerung eingeführt wurden und Oz zudem besonders benachteiligt ist, wenn viele Propagierer verwendet werden (wegen des Kopierens von Berechnungsräumen bei Distribuierung; siehe Kapitel 7). Doch selbst wenn zwei Systeme Propagierer für den gleichen Constraint enthalten, kann sich die Propagierung stark unterscheiden. Bei kommerziellen Systemen wie ILOG SOLVER ist aber praktisch nichts über die konkrete Implementierung eines Propagierers bekannt. Somit kann ein Vergleich anhand von kleinen Beispielproblemen hauptsächlich nur untersuchen, inwieweit die Systeme expressiv sind, um die Probleme auszudrücken und ob sie prinzipiell effizient genug sind, um die getesteten Probleme zu lösen. Zudem beleuchten die hier gezeigten Beispiele nur einen kleinen Teil der Funktionalität eines FD-Systems. Für Probleme aus der Praxis zeigen gerade die Kapitel 8, 9 und 10 sowie der folgende Abschnitt 13.3 die Expressivität und die Effizienz von Oz.

### 13.2.1 Verglichene Systeme

Es werden die folgenden FD-Systeme verglichen.

- DFKI Oz, Version 2.04
- `clp(FD)`, Version 2.21
- AKL(FD), Version 1.0[1]
- ECL$^i$PS$^e$, Version 3.5.2
- SICStus, Version 3#5 (Entwicklungsversion vom August 1997)[2]
- ILOG SOLVER, Version 3.2
- CHIP, Version 5.0.1[3]

[1] Dies ist die Version, die von Björn Carlson für seine Dissertation [Car95] verwendet wurde und von ihm freundlicherweise zu Experimentierzwecken zur Verfügung gestellt wurde.

[2] Freundlicherweise von Mats Carlsson zur Verfügung gestellt. Die Effizienz der Entwicklungsversion liegt teilweise deutlich über der Effizienz der Version 3#5.

[3] Freundlicherweise wurde uns die CHIP-Lizenz von Alexander Bockmayr zurVerfügung gestellt.

### 13.2.2 Verwendete Rechnerkonfigurationen

Da nicht alle FD-Systeme auf den gleichen Betriebssystemen verfügbar waren, verwendeten wir eine Sun SPARC ELC und zwei Sun SPARC20. Auf der ELC, 33 MHz, 32 MB Hauptspeicher, 109 MB virtuellem Speicher und dem Betriebssystem SunOS 4.1.4 liefen Oz, `clp(FD)`, AKL(FD) und ECL$^i$PS$^e$. Auf einer Sun SPARC20, 70MHz, 192 MB Hauptspeicher, 242 MB virtuellem Speicher, 2 SuperSPARC-Prozessoren und dem Betriebssystem SunOS 5.5.1 liefen Oz, SICStus und ILOG SOLVER. Auf einer Sun SPARC20, 125 MHz, 448 MB Hauptspeicher, 1.1 GB virtuellem Speicher, 2 HyperSPARC Prozessoren und dem Betriebssystem SunOS 5.5.1 liefen Oz und CHIP.

### 13.2.3 Probleme

Es wurden die folgenden Beispielprobleme verwendet. Alle verwendeten Programme sind unter [Wür98] verfügbar.

- Für *queens16*, *queens100* und *queens450* sind $n$ Damen auf einem $n \times n$ Schachbrett so anzuordnen, daß sich keine zwei Damen gegenseitig schlagen können (siehe Abschnitt 3.1.2).

- *alpha* ist ein krypto-arithmetisches Problem mit 26 FD-Variablen mit jeweils dem Bereich 1#26, 20 linearen Gleichungen und dem Constraint, daß die Variablen paarweise verschiedene Werte annehmen müssen.

- *eq20* ist ein System aus 20 linearen Gleichungen über 7 FD-Variablen mit jeweils dem Bereich 0#10.

- In *magic100* und *magic300* wird eine Folge von $n$ Zahlen berechnet, so daß jede davon die Zahl der Vorkommen ihrer Position in der Folge angibt (siehe Abschnitt 3.1.3).

- In *squares* ist ein Quadrat der Seitenlänge 112 mit 21 kleineren vorgegebenen Quadraten vollständig aufzufüllen [AB93].

- Die Probleme *incons1* bis *incons4* bestehen jeweils aus einer Konjunktion von Constraints, für die Propagierung alleine die Inkonsistenz zeigen sollte. Das Problem *incons1* ist der Constraint $x < y \wedge y < x$ mit den Bereichen $0\#10^6$ für $x$ und $y$. Das Problem *incons2* ist der Constraint

  $$a+b+c+1 = d \wedge d+e+f+1 = g \wedge g+h+i+1 = j \wedge j+k+l+1 = a$$

  mit den Bereichen $0\#10^6$ für die vorkommenden Variablen. In *incons3* wird ein "Ring" von 20 Ungleichungen der Art $x_i < x_{i+1}$ und in *incons4* ein "Ring" von 20 Gleichungen der Art $x_i + x_{i+1} + x_{i+2} + x_{i+3} + 1 = x_{i+4}$ aufgebaut. Damit ist gemeint, daß der 20. Constraint $x_{20} < x_1$ bzw. $x_{77} + x_{78} + x_{79} + x_{80} = x_1$ ist, so daß die Konjunktion der Constraints unerfüllbar ist. Der Bereich einer vorkommenden Variablen ist jeweils wieder $0\#10^6$.

### 13.2.4 Ergebnisse

Die Ergebnisse auf der SPARC ELC sind in Tabelle 13.1 und die auf den zwei SPARC20 in Tabelle 13.2 wiedergegeben. Tabelle 13.3 gibt das Verhältnis der Laufzeiten wieder.

Für Oz beinhalten die Laufzeiten auch die Zeiten für Kopieren von Berechnungsräumen. Die Zahlen in Klammern für Oz geben den prozentualen Anteil des Kopierens von Berechnungsräumen an der gesamten Laufzeit an. Die ersten Zahlen für AKL(FD) beinhalten Kopierzeiten, während die zweiten Zahlen für AKL(FD) die Laufzeit angeben, wenn Kopieren durch sogenanntes Trailing ersetzt wird (siehe [Car95]; da diese Implementierung aber nur für reine Indexical-Programme realisiert wurde, werden in Tabelle 13.3 diese Zahlen nicht in Relation zu Oz gesetzt). Die Laufzeiten für Oz, AKL(FD), ECL$^i$PS$^e$ und SICStus enthalten nicht die Zeiten für die Speicherbereinigung, da diese problemspezifisch stark von den jeweils gesetzten Systemparametern abhängen. Die Laufzeiten für CHIP wurden auf Laufzeiten für diejenige SPARC20 umgerechnet, auf der u. a. SICStus lief. In Tabelle 13.1 bedeutet der Eintrag †, daß bei der Programmausführung ein Fehler aufgetreten ist. Der Eintrag $>$ 1000 bedeutet, daß das Programm nach 1 000 Sekunden ergebnislos abgebrochen wurde.

**Tabelle 13.1** Laufzeitergebnisse auf einer SPARC ELC

| Problem | Oz | AKL(FD) | `clp(FD)` | ECL$^i$PS$^e$ |
|---|---|---|---|---|
| queens16 | 6.58 (28%) | 12.06/3.75 | 1.52 | 13.62 |
| queens100 | 0.95 (48%) | $>$ 1000/248.88 | 79.76 (1.02) | 662.28 (5.22) |
| queens450 | 29.02 (46%) | $>$ 1000/$>$ 1000 | † | $>$ 1000 |
| alpha | 29.53 (33%) | 189.12/28.67 | 12.65 | 242.63 |
| eq20 | 0.48 (14%) | 2.54/1.62 | 0.18 | 1.05 |
| magic100 | 12.72 (26%) | 139.73/149.46 | 42.33 | 201.64 |
| magic300 | 195.96 (32%) | †/† | † | $>$ 1000 |
| squares | 5.03 (22%) | †/† | 9.42 | 84.1 |
| incons1 | 18.48 | † | 5.59 | 149.97 |
| incons2 | 135.68 | 187.63 | 71.75 | 548.85 |
| incons3 | 35.37 | † | 5.63 | $>$ 1000 |
| incons4 | 219.20 | 455.39 | 102.73 | $>$ 1000 |

Im folgenden machen wir für die verwendeten Beispiele einige Anmerkungen zu den konkreten Programmen.

- Für *queens16* etc. wird in Oz, ILOG SOLVER und CHIP eine Modellierung mit drei globalen Constraints verwendet (siehe Abschnitt 3.1.2). Die anderen Systeme verwenden $O(n^2)$ viele Propagierer, da eine Modellierung mit globalen Constraints dort nicht möglich ist. Für *queens16* wird die naive Distribuierungsstrategie verwendet (für eine Liste von Variablen wird jeweils die am weitesten links stehende Variable in der Liste zuerst determiniert). Für die anderen Probleme wird first-fail verwendet. Für ILOG SOLVER, SICStus und Oz ergibt sich hierfür die gleiche Anzahl von Wahlpunkten. ECL$^i$PS$^e$, AKL(FD), `clp(FD)` und CHIP unterstützen eine von Oz verschiedene first-fail-Distribuierungsstrategie (die Reihenfolge der Variablen in der Liste der zu determinierenden Variablen wird in CHIP etc. verändert). Jedoch ist für ECL$^i$PS$^e$, `clp(FD)` und CHIP in Klammern die Laufzeit

angegeben, wenn man die gleiche Strategie wie in Oz verwendet.[4] In AKL(FD) stand leider die Funktionalität zur Bereichsreflektion nicht zur Verfügung, um diese Strategie zu implementieren.

Hier sieht man gut den Vorteil von globalen Constraints, ohne die der Speicherverbrauch und die Laufzeit drastisch ansteigen. Verwendet man in Oz ein Programm mit $O(n^2)$ Propagierern für $n = 100$ und first-fail, so ist der Anteil von Kopieren und Speicherbereinigung an der Gesamtlaufzeit etwa 90%. Rechnet man die Speicherbereinigung heraus, so ergibt sich auf der SPARC20 eine Laufzeit von etwa 6.23 Sekunden, wobei 85% der Zeit für Kopieren verbraucht werden. Der Speicherverbrauch beträgt 28.6 MB gegenüber 0.76 MB für die Variante mit globalen Constraints. Durch den Einsatz von Wiederberechnung (engl. recomputation) der Tiefe 10 (Abschnitt 10.4) kann die Laufzeit auf 1.55 Sekunden (40% Kopieren) verkürzt werden. Der Speicherverbrauch beträgt dann nur noch 4.8 MB. *queens450* kann von `clp(FD)` nicht gelöst werden, da es hier zu einem Speicherallozierungsfehler kommt (für $n = 300$ werden in `clp(FD)` mit der gleichen Distribuierungsstrategie wie in Oz 13.04 Sekunden benötigt; in Oz 1.61 Sekunden).

- Für *alpha* und *eq20* wurde die naive Distribuierungsstrategie verwendet.

- Auch in *magic100* und *magic300* wurde die naive Distribuierungsstrategie verwendet. `clp(FD)`, AKL(FD) und ECL$^i$PS$^e$ verfügen für dieses Problem über keine globalen Constraints. In Oz wird der globale Constraint `FD.exactly` benutzt (CHIP, SICStus und ILOG SOLVER verwenden entsprechende Constraints). Bei *magic300* gibt es bei AKL(FD) einen Systemabsturz, `clp(FD)` führt zu einem Speicherallozierungsfehler und ECL$^i$PS$^e$ wurde nach 1 000 Sekunden abgebrochen. Der Eintrag für ILOG SOLVER gibt die Laufzeit an, wenn man das gleiche Modell wie in Oz verwendet. Da es in ILOG SOLVER jedoch auch die Möglichkeit gibt, alle verwendeten globalen Constraints durch einen einzigen Propagierer zu realisieren, ist die entsprechende Laufzeit in Klammern angegeben.

- Für das Problem *squares* wird eine spezielle Distribuierungsstrategie benutzt (siehe [AB93, VSD95]). Oz und SICStus verwenden redundante globale Constraints (Kapazitätsconstraints für kumulative Schedulingprobleme). Außerdem wird in Oz ein globaler Constraint (`FD.distinct2`) verwendet, der das Überlappen von Rechtecken verhindert. Für AKL(FD) führte dieses Problem zu Speicherproblemen (die Trailing-Variante kann mit dieser Distribuierungsstrategie nicht verwendet werden). In dem ILOG-Programm werden globale Constraints aus der Bibliothek ILOG SCHEDULER, Version 2.2, verwendet. In CHIP werden die gleichen globalen Constraints wie in Oz verwendet (jedoch mit unterschiedlichem Propagierungsverhalten).

- Die Probleme *incons1* bis *incons4* zeigen recht gut die Effizienz der verwendeten Propagierer und der zugrundeliegenden Implementierung des FD-Systems. In Oz wurden die Probleme in einem Berechnungsraum ausgeführt, um lokale Propagierung ausnutzen zu können (siehe Abschnitt 11.5.2). AKL(FD) resultierte in einem Fehler für *incons1* und *incons3*, da die Probleme fälschlicherweise gelöst wurden. Beim Übergang von *incons1* zu *incons3* ist ein Anstieg der Rechenzeit für Oz, CHIP und ILOG SOLVER zu beobachten. Für Oz liegt das daran, daß zusätzlich zu der sehr einfachen Propagierung jedesmal ein

[4] Die Strategie wurde von Antonio Fernandez zur Verfügung gestellt, der sie wiederum von Joachim Schimpf erhalten hat.

Gleichheitstest und ein Test auf Subsumtion gemacht werden muß (vermutlich gilt gleiches auch für CHIP und ILOG SOLVER). `clp(FD)` ist hier sehr schnell, da es spezielle Bibliotheks-Indexicals gibt (für Gleichungen bzw. Ungleichungen ohne Koeffizienten), in die die Constraints übersetzt werden.

**Tabelle 13.2** Laufzeitergebnisse auf einer SPARC20

| Problem | Oz | CHIP | SICStus | ILOG SOLVER |
|---|---|---|---|---|
| queens16 | 1.38 (28%) | 0.26 | 0.64 | 0.75 |
| queens100 | 0.19 (48%) | 20.16 (0.85) | 0.39 | 0.22 |
| queens450 | 7.11 (46%) | 8.73 (31.82) | 27.17 | 9.12 |
| alpha | 6.11 (33%) | 17.02 | 6.91 | 1.77 |
| eq20 | 0.10 (14%) | 0.09 | 0.09 | 0.04 |
| magic100 | 2.61 (26%) | 18.64 | 3.50 | 2.05 (1.11) |
| magic300 | 42.03 (32%) | 699.0 | 55.98 | 72.44 (21.68) |
| squares | 1.01 (23%) | 0.18 | 3.03 | 0.45 |
| incons1 | 3.86 | 2.46 | 5.81 | 3.10 |
| incons2 | 27.31 | 41.78 | 45.95 | 17.81 |
| incons3 | 7.22 | 3.87 | 5.66 | 4.87 |
| incons4 | 45.27 | 61.85 | 66.93 | 27.46 |

**Tabelle 13.3** Laufzeitvergleich

| Problem | *AKL(FD)* / Oz | `clp(FD)` / Oz | Ecl$^i$ps$^e$ / Oz | *CHIP* / Oz | *SICStus* / Oz | *IlogSolver* / Oz |
|---|---|---|---|---|---|---|
| queens16 | 1.83 | 0.23 | 2.07 | 0.19 | 0.46 | 0.54 |
| queens100 | - | 83.95 (1.07) | 697.14 (5.49) | 106.11 (4.47) | 2.05 | 1.16 |
| queens450 | - | - | - | 1.12 (4.48) | 3.82 | 1.28 |
| alpha | 6.40 | 0.43 | 8.22 | 2.79 | 1.13 | 0.29 |
| eq20 | 5.29 | 0.38 | 2.19 | 0.90 | 0.90 | 0.40 |
| magic100 | 10.99 | 3.33 | 15.85 | 7.14 | 1.34 | 0.79 (0.43) |
| magic300 | - | - | - | 16.63 | 1.33 | 1.72 (0.52) |
| squares | - | 1.87 | 16.72 | 0.18 | 3.00 | 0.45 |
| incons1 | - | 0.30 | 8.12 | 0.64 | 1.51 | 0.80 |
| incons2 | 1.38 | 0.53 | 4.05 | 1.53 | 1.68 | 0.65 |
| incons3 | - | 0.16 | - | 0.54 | 0.78 | 0.67 |
| incons4 | 2.08 | 0.47 | - | 1.37 | 1.48 | 0.61 |

### 13.2.5 Bewertung

Wir bewerten die Expressivität und die Effizienz der verglichenen Systeme. Dabei schließen wir AKL(FD) aus, da nur ein experimenteller Prototyp existiert.

`clp(FD)` ist ein System, dem es vor allem an globalen Constraints mangelt (neben der mangelnden Expressivität für komplexe Anwendungen aus der Praxis). Auch ECL$^i$PS$^e$ leidet unter

fehlenden globalen Constraints. Die übrigen Systeme bieten für die betrachteten Probleme ausreichende Expressivität.

ECL$^i$PS$^e$ ist für die betrachteten Probleme sehr ineffizient. Die übrigen Systeme sind in etwa in ihrer Effizienz miteinander vergleichbar.

Weitere Schlüsse sind aufgrund der begrenzten Aussagekraft solcher Tests nicht zulässig (siehe auch die Vorbemerkung zu diesem Abschnitt).

## 13.3 Evaluierung der Schedulingtechniken von Oz

Um die Effizienz von Oz zum Lösen von Schedulingproblemen zu unterstreichen, wurden die Schedulingtechniken von Oz, Version 2.0.4, mit denen von ILOG SCHEDULER [ILOG96a], Version 2.2, und CLAIRE [CL94b], Version 1.040, anhand einer Reihe von Standardproblemen verglichen (siehe Abschnitt 9.5 für eine Beschreibung der Systeme). Dieser Vergleich wurde sowohl für den Beweis der Optimalität einer Lösung als auch für das Finden guter Lösungen durchgeführt. ILOG SCHEDULER und CLAIRE wurden ausgewählt, da sie beide den *state-of-the-art* für das Lösen von Job-Shop-Problemen (siehe Abschnitt 9.3.3) repräsentieren. CHIP wurde nicht herangezogen, da in der Distribution [COS96] keine Serialisierer zur Lösung von Job-Shop-Problemen vorhanden sind, ohne die solche Probleme nicht effizient gelöst werden können. Alle Messungen wurden mit der gleichen Rechnerkonfiguration wie in Abschnitt 9.4 durchgeführt. Abschnitt 9.4 enthält auch die Beschreibung der hier betrachteten Probleme und wie die Laufzeit in Oz gemessen wurde (insbesondere ist die Zeit für Speicherbereinigung enthalten). Für Oz wird immer `FD.schedule.serialized` [HMSW97] als Propagierer für Kapazitätsconstraints (dem Propagierer liegt der Zwei-Phasen-Algorithmus aus Abschnitt 5.2.4 zugrunde) und einer der beiden aufgabenorientierten Serialisierer aus Abschnitt 6.3 verwendet (danach unterschieden, ob eine gute Lösung gefunden oder ein Optimalitätsbeweis geführt werden soll).Wir geben die Anzahl der Fehlerknoten im Vergleich an, da diese Größe bei allen drei Systemen extrahiert werden konnte.

Da in den jeweiligen Systemen sowohl unterschiedliche Kapazitätsconstraints als auch unterschiedliche Distribuierungsstrategien verwendet werden, läßt dieser Vergleich nur bedingt eine Wertung zu (die Techniken können prinzipiell in allen Systemen implementiert werden). Der Vergleich belegt hauptsächlich, daß Oz sowohl kompetitiv zu kommerziellen Schedulingsystemen (ILOG SCHEDULER) wie auch zu speziell für komplexe Optimierungsprobleme entwickelten Systemen (CLAIRE) ist. Schedulingtechniken können in Oz so effizient implementiert werden, wie in den verglichenen Systemen.

### 13.3.1 Optimalitätsbeweis

In Tabelle 13.4 werden Schedulingprogramme zum Optimalitätsbeweis in Oz, ILOG SCHEDULER und CLAIRE verglichen. Etwas überraschend ist, daß mit Oz teilweise sogar weniger Fehlerknoten benötigt werden als mit CLAIRE, obwohl der Kapazitätsconstraint in Oz weniger stark propagiert und der in Oz verwendete Serialisierer eine konzeptuelle Vereinfachung der CLAIRE-Version ist.

In Oz wird der aufgabenorientierte Serialisierer für einen Optimalitätsbeweis eingesetzt (`FD.schedule.taskIntervalsDistP`). Für ILOG SCHEDULER wurde eine ressourcenorientierte Distribuierungsstrategie und eine stark propagierende Version des Edge-Findings aus [Nui94] gewählt.[5] Für CLAIRE wurden Aufgabenintervalle und der speziell für Optimalitätsbeweise angepaßte Serialisierer verwendet. Obwohl in CLAIRE Aufgabenintervalle verwendet werden, sind die Schedulingprogramme recht effizient. Dies liegt an der selbst schon sehr effizienten Implementierung von Aufgabenintervallen (auf die sowohl Propagierer als auch Serialisierer Zugriff haben) und der Möglichkeit in CLAIRE, diese inkrementell zu aktualisieren (siehe auch Abschnitt 7.3.4).

Die Resultate für CLAIRE verbessern die Resultate, die in [CL95] angegeben wurden. Schon die Ergebnisse in [CL95] sind bzgl. der notwendigen Fehlerknoten vergleichbar mit den derzeit besten Resultaten aus dem Operations Research (siehe zum Beispiel [CP94]) und um eine Größenordnung besser als die Resultate in [AC91].

**Tabelle 13.4** Oz im Vergleich zu ILOG SCHEDULER und CLAIRE, Optimalitätsbeweis

| | Oz | | ILOG | | CLAIRE | | ILOG/Oz | | CLAIRE/Oz | |
|---|---|---|---|---|---|---|---|---|---|---|
| Problem | Fails | CPU | Fails | CPU | Fails | CPU | Fails | CPU | Fails | CPU |
| MT10 | 1 799 | 28.7 | 5 853 | 69.7 | 1 569 | 27.5 | 3.25 | 2.43 | 0.87 | 0.96 |
| ABZ5 | 1 431 | 23.8 | 2 548 | 23.4 | 1 349 | 21.0 | 1.78 | 0.98 | 0.94 | 0.88 |
| ABZ6 | 148 | 2.3 | 207 | 2.3 | 157 | 2.6 | 1.40 | 1.00 | 1.06 | 1.13 |
| La19 | 1 066 | 17.7 | 3 786 | 35.1 | 1 067 | 15.6 | 3.55 | 1.93 | 1.00 | 0.88 |
| La20 | 881 | 13.9 | 10 384 | 72.2 | 905 | 13.0 | 11.79 | 5.19 | 1.03 | 0.94 |
| ORB1 | 7 528 | 120.9 | 3 925 | 42.9 | 7 327 | 124.8 | 0.52 | 0.35 | 0.97 | 1.03 |
| ORB2 | 425 | 7.2 | 16 922 | 183.0 | 453 | 7.9 | 39.82 | 25.42 | 1.07 | 1.10 |
| ORB3 | 22 579 | 339.4 | 16 845 | 211.3 | 32 505 | 525.2 | 0.75 | 0.62 | 1.44 | 1.55 |
| ORB4 | 1 034 | 15.7 | 17 677 | 207.6 | 1 060 | 16.3 | 17.10 | 13.22 | 1.03 | 1.04 |
| ORB5 | 869 | 14.4 | 3 031 | 27.2 | 795 | 13.9 | 3.49 | 1.89 | 0.91 | 0.97 |

### 13.3.2 Untere Schranken

Verwendet man in Oz (mit dem Propagierer `FD.schedule.serialized`) die in Abschnitt 9.3.5 beschriebene Technik zur Bestimmung einer unteren Schranke der Schedulelänge, indem auf das Fehlschlagen der Propagierung bei einer vorgegebenen Schedulelänge gewartet wird, so erhält man bis auf drei Ausnahmen die gleichen unteren Schranken wie in [CL95] mit Aufgabenintervallen. Bei den Ausnahmen ist die Schranke in Oz um etwa 6 Promille der optimalen Schedulelänge kleiner als die in [CL95] gefundene untere Schranke. Für alle Probleme zusammen sind die Schranken im Mittel etwa 8% vom Optimum entfernt und man benötigt zur Berechnung weniger als eine Sekunde Rechenzeit. Die unteren Schranken mit der Operations-Research-Methode aus [AC91] sind im Schnitt 11% vom Optimum entfernt und damit schlechter.

[5] Genauer wurde für die Ressourcenauswahl `IlcSelResMinLocalSlack`, für die Aufgabenauswahl `IlcSelFirstRCMinStartMax` und `setEdgeFinder` mit dem Parameter Zwei für den Kapazitätsconstraint verwendet [ILOG96a]. Diese Kombination brachte die besten Ergebnisse in Bezug auf Fehlerknoten und Laufzeit. Die Strategie `IlcSelLastRCMinStartMax` des Benutzerhandbuchs war leider nicht funktionsfähig.

### 13.3.3 Optimierung

Tabelle 13.5 dient dem Vergleich der Optimierungsfähigkeiten von Oz, ILOG SCHEDULER und CLAIRE. In der Spalte *Oz* werden für die Optimierungsphase die Parametereinstellungen aus Abschnitt 9.3.3 verwendet (als Serialisierer wird der aufgabenorientierte Serialisierer für das Finden guter Lösungen verwendet: `FD.schedule.taskIntervalsDist0`). Nachdem die Optimierungsphase beendet ist, wird die Beweisphase mit dem zusätzlichen Constraint gestartet, daß die Schedulelänge um mindestens Eins kleiner sein muß als die Lösung, die in der Optimierungsphase gefunden wurde. Die Beweisphase läuft solange, bis die Optimalität der zuletzt gefundenen Lösung bewiesen ist. Die Optimierungsphase in Oz findet mit den vorgegebenen Rechenressourcen (siehe Abschnitt 9.3.3) sechs von zehn optimalen Lösungen. Die besten gefundenen Lösungen für ABZ5, ORB2, ORB4 und ORB5 sind jedoch jeweils nur 4, 1, 8 und 2 Promille von der optimalen Lösung entfernt (siehe auch Abbildung 9.8). Die Spalte ILOG enthält die Daten für eine in [BPN95a, Nui94] vorgestellte Strategie, die die optimale Lösung für die betrachteten Probleme findet und die Optimalität beweist (auch hier ist die Optimierungsphase, die auf einer ressourcenbeschränkten Form von Branch-and-Bound beruht, unvollständig). Da diese Strategie nicht in der ILOG-Distribution verfügbar ist, sind die Laufzeiten in [BPN95a] (auf einer IBM RS6000) in Laufzeiten auf der hier verwendeten SPARC20 umgerechnet worden. Eine entsprechende Funktionalität ist in der CLAIRE-Distribution nicht vorhanden.

Die Spalte *Oz(opt)* in Tabelle 13.5 enthält die Resultate der Optimierungsphase von Oz, wenn die Parameter (siehe Abschnitt 9.3.3) `MaxFailures` auf 2 430 und `Iterations` auf Vier gesetzt werden und die Suche bei Erreichen der optimalen Lösung abgebrochen wird (es werden mit den erweiterten Ressourcen also alle optimalen Lösungen in der Optimierungsphase gefunden). Die mit dem gierigen (engl. greedy) Algorithmus (siehe Abschnitt 9.3.3) gefundene Lösung ist im Schnitt nur 11% vom Optimum entfernt. Beachte, daß die Zahlen in dieser Spalte nicht in Widerspruch zur Spalte *Oz* stehen, da dort nach Ende der Optimierungsphase die Beweisphase gestartet wird, die nicht von einer partiellen Lösung (wie bei dem Shuffle-Algorithmus) ausgehen muß. Die Spalte CLAIRE enthält die Daten für die Optimierungsphase für CLAIRE, wobei die Suche auch nach Erreichen der optimalen Lösung abgebrochen wird. Die teilweise besseren Ergebnisse mit CLAIRE beruhen darauf, daß hier unter anderem weitere Shuffle-Techniken verwendet werden [CL95]. In der Distribution von ILOG SCHEDULER war eine entsprechende Funktionalität nicht vorhanden. Diese Tabelle zeigt, daß Oz auch für das Finden guter Lösungen kompetitiv zu speziellen Systemen zum Lösen von Schedulingproblemen ist. Die mit Oz erzielten Ergebnisse haben eine bessere Qualität als die in [AC91] erzielten Resultate.

### 13.3.4 Schwere Probleme

Es stellt sich die Frage, ob der Schedulingansatz in Oz auch für größere und schwierigere Probleme trägt. In [VAL94] wird eine Untersuchung für als besonders schwierig betrachtete Probleme vorgestellt (mit Ausnahme von MT10 stammen diese aus [Law84]). Die erfolgreichste Strategie in [VAL94] (in Bezug auf Rechenzeit und Qualität der Lösungen) ist eine spezielle Tabu-Suche [NS93]. Ansätze aus dem Simulated Annealing oder Genetische Algorithmen schneiden bzgl. der gefundenen oberen Schranken für die untersuchten Probleme sehr viel schlechter ab. Der constraintbasierte Ansatz in [NA96] schneidet etwa so gut ab wie Simulated Annealing, ist jedoch sehr viel langsamer. Die in [CL95] vorgestellte Methode liefert bessere obere Schranken

**Tabelle 13.5** Oz im Vergleich zu ILOG SCHEDULERund CLAIRE, Optimierung

| | Oz | | ILOG | | Oz(opt) | | CLAIRE | |
|---|---|---|---|---|---|---|---|---|
| Problem | Fails | CPU | Fails | CPU* | Fails | CPU | Fails | CPU |
| MT10 | 3 322 | 68.1 | 13 684 | 113.4 | 412 | 20.0 | 651 | 22.7 |
| ABZ5 | 3 506 | 73.7 | 19 303 | 135.6 | 11 711 | 208.7 | 9 946 | 196.8 |
| ABZ6 | 1 241 | 30.2 | 6 227 | 48.4 | 56 | 11.0 | 250 | 15.1 |
| La19 | 2 467 | 56.2 | 18 102 | 129.6 | 103 | 13.0 | 102 | 6.9 |
| La20 | 2 146 | 45.7 | 40 597 | 238.8 | 19 | 9.8 | 400 | 17.2 |
| ORB1 | 9 520 | 173.2 | 22 725 | 195.8 | 630 | 29.1 | 796 | 30.1 |
| ORB2 | 2 703 | 57.9 | 31 490 | 243.8 | 4 795 | 93.6 | 185 | 9.2 |
| ORB3 | 24 160 | 388.0 | 36 729 | 291.4 | 489 | 30.3 | 835 | 34.6 |
| ORB4 | 3 831 | 71.8 | 13 751 | 102.7 | 11 256 | 173.9 | 4 864 | 110.8 |
| ORB5 | 3 108 | 62.9 | 12 648 | 101.4 | 3 849 | 75.6 | 2 348 | 61.0 |

als der Tabu-Ansatz, benötigt aber mehr Rechenzeit als dieser.

In Tabelle 13.6 wird für die in [VAL94] betrachteten Probleme die Optimierungsphase von Oz getestet. Auch hier wird die Suche abgebrochen, nachdem eine Lösung mit einer vorgegebenen Schedulelänge (in der Spalte *Best*) gefunden wurde (um einen Vergleich mit CLAIRE zu ermöglichen). Die Spalte *Optimum* gibt die optimale Schedulelänge und *Size* die Problemgröße $n \times m$ wieder (zur Erinnerung: $n$ ist die Anzahl der Jobs und $m$ die Anzahl der Ressourcen). Für das Problem LA29 liegt das noch unbekannte Optimum zwischen 1 130 und 1 157. Für die Spalte *Oz* wurden die Parametereinstellungen aus Abschnitt 9.3.3 verwendet (nur für LA27 wurde der Parameter `MaxFailures` auf 2 430 erhöht). Auch für diese Probleme ist Oz kompetitiv. Verwendet man für die Probleme LA24, LA25 und LA29 den Serialisierer `FD.schedule.taskIntervalsDistP`, so findet man Lösungen mit einer Länge von jeweils 935, 977 und 1 173. Dabei werden die Parameter `MaxFailures` auf 7 290 und `Iterations` auf 5 gesetzt.

**Tabelle 13.6** Vergleich der Optimierungsfähigkeiten für sehr schwierige Probleme

| | Size | Optimum | CLAIRE | | | Oz | | |
|---|---|---|---|---|---|---|---|---|
| Problem | | | Best | Fails | CPU | Best | Fails | CPU |
| MT10 | 10×10 | 930 | 930 | 651 | 22.7 | 930 | 412 | 20.0 |
| LA02 | 10×5 | 655 | 655 | 13 | 1.0 | 655 | 33 | 2.9 |
| LA19 | 10×10 | 842 | 842 | 102 | 6.3 | 842 | 103 | 13.0 |
| LA21 | 15×10 | 1 046 | 1 046 | 544 | 94.9 | 1 046 | 2 426 | 179.2 |
| LA24 | 15×10 | 935 | 935 | 1 276 | 136.8 | 939 | 1 767 | 119.9 |
| LA25 | 15×10 | 977 | 977 | 172 | 76.1 | 979 | 2 325 | 155.5 |
| LA27 | 20×10 | 1 235 | 1 235 | 4 001 | 1 012.2 | 1 240 | 59 000 | 3 862.6 |
| LA29 | 20×10 | 1 130-1157 | 1 168 | 4 408 | 1 033.4 | 1 191 | 3 612 | 715.8 |
| LA36 | 15×15 | 1 268 | 1 268 | 1 337 | 344.8 | 1 268 | 2 792 | 266.0 |
| LA37 | 15×15 | 1 397 | 1 397 | 838 | 268.5 | 1 397 | 1 380 | 213.0 |
| LA38 | 15×15 | 1 196 | 1 196 | 2884 | 473.3 | 1 196 | 4 302 | 409.8 |
| LA39 | 15×15 | 1 233 | 1 233 | 630 | 198.1 | 1 233 | 1 021 | 172.3 |
| LA40 | 15×15 | 1 222 | 1 222 | 19 424 | 1 753.5 | 1 222 | 2 294 | 307.6 |

In [Zho97] wird ein spezieller Propagierer zur Lösung von Job-Shop-Problemen verwendet, der eine Menge von FD-Variablen sortiert (siehe auch [BC97]). Mit Hilfe dieses Propagierers werden

auch einige der als sehr schwer angesehenen Probleme aus [VAL94] angegangen. In Tabelle 13.7 werden die Ergebnisse aus [Zho97] mit den Resultaten für Oz verglichen (das System in [Zho97] war nicht zu Experimentierzwecken verfügbar). Spalte *CPU(h)* enthält die Laufzeit in Stunden. Während für das schwerste Problem LA21 mit Oz etwa sieben Stunden benötigt werden, benötigt Zhou mehrere Tage. Die Optimalität der Lösungen der Probleme LA21 und LA38 wurde erst 1995 bewiesen [CL95, MS96]. In [MS96] wird ein spezieller in C entwickelter Algorithmus zum Finden von unteren Schranken verwendet (es wird die sogenannte *shaving*-Technik benutzt). Da in [MS96] für LA21 und LA38 jeweils mehr als drei Tage Rechenzeit benötigt wird, ist Oz somit zur Zeit das System, das diese Probleme am effizientesten löst.

**Tabelle 13.7** Optimalitätsbeweis für sehr schwierige Probleme

| | [Zho97] | CLAIRE | | Oz | |
|---|---|---|---|---|---|
| Problem | Fails | Fails | CPU(h) | Fails | CPU(h) |
| LA21 | 2 473 624 | 1 287 337 | 8.60 | 852 597 | 7.13 |
| LA24 | 260 409 | 749 794 | 7.02 | 412 178 | 3.57 |
| LA36 | 1 676 | 34 999 | 0.30 | 54 941 | 0.65 |
| LA38 | 783 648 | 356 908 | 5.17 | 241 996 | 3.04 |
| LA39 | 881 | 7 227 | 0.05 | 3 112 | 0.04 |
| LA40 | 55 030 | 14 452 | 0.22 | 12 904 | 0.17 |

### 13.3.5 Zusammenfassende Bewertung

Die Evaluierung der in Oz implementierten Schedulingtechniken zeigt deutlich die Effizienz des Systems. Dies gilt sowohl für Optimalitätsbeweise als auch für das Finden guter bzw. optimaler Lösungen. Selbst als sehr schwierig eingestufte Probleme können effizient gelöst werden. Die Effizienz von Oz zum Lösen der beschriebenen Probleme ist vergleichbar mit den zur Zeit besten Verfahren aus dem Operations Research [AC91, CP94, Nui94, VAL94] und der Constraintprogrammierung [CL95, CL94b, BPN95a, ILOG96a].

# Kapitel 14

# Schluß

## 14.1 Beitrag

Ausgangspunkt dieser Arbeit war die Frage, ob kombinatorische Probleme in der nebenläufigen Constraintsprache Oz effizient gelöst werden können. Diese Frage ist, zumindest für die untersuchten Probleme, positiv zu beantworten. Dies gilt einmal für Standardprobleme, wie sie in Vergleichen zwischen Constraintsystemen üblich sind (Abschnitt 13.2). Zum anderen, und das ist unserer Meinung nach viel überzeugender, konnten wir mit Oz eine Reihe von großen und schwierigen Problemen effizient lösen. Dies umfaßt sowohl akademische Schedulingprobleme, die sonst nur von Spezialsystemen gelöst werden (siehe Kapitel 9 und Abschnitt 13.3), als auch Probleme aus der Praxis (die Fallstudien in Kapitel 8 und 10).

Bei den Fallstudien konnten wir die Expressivität der nebenläufigen Constraintsprache Oz ausnutzen (wobei wir selbst nur einen Teil zur Expressivität der Sprache beigesteuert haben, nämlich den Propagierungsteil des FD-Systems). Bei allen Problemen war es notwendig, möglichst schnell und einfach viele Propagierer und Distribuierer zu testen, um die beste Kombination herauszufinden. Hier konnten wir sowohl die Vorteile von Oz als eine Hochsprache als auch die strikte Trennung von Proplemrepräsentation und Lösungssuche in der Constraintprogrammierung voll ausnutzen. In den Fallstudien in Kapitel 8 und 10 mußten wir neue Distribuierer programmieren, die in Oz nicht vordefiniert sind. Zur Realisierung des Oz-Schedulers konnten wir sowohl von der Möglichkeit profitieren, neue Suchstrategien in Oz zu programmieren (in einem Prototyp), als auch heuristische Verfahren mit Constraintprogrammierung in Oz zu verbinden. Hier waren auch höhere Prozeduren hilfreich, die uns das Implementieren von Scheduling-Compilern ermöglichten. Bei allen Fallstudien konnten die realisierten Prototypen von der objekt-orientierten Programmierung in Oz profitieren.

Im folgenden fassen wir noch einmal die von uns geleisteten Beiträge zusammen.

Wir haben ein sprachunabhängiges formales Modell zum constraintbasierten Lösen kombinatorischer Probleme vorgestellt, das auf Propagierung und Distribuierung beruht. Wir haben zur Realisierung von Kapazitätsconstraints und von Serialisierern einige der derzeit besten Schedulingtechniken (Edge-Finding) aus dem Operations Research für disjunktive und kumulative Schedulingprobleme [CP89, AC91, MS96, CL94a, BPN95b, Nui94] im Rahmen unseres Mo-

dells formuliert (Kapitel 5 und 6). Insbesondere haben wir den Zwei-Phasen-Algorithmus in die Constraintprogrammierung übertragen und verstärkt, sowie auf kumulative Schedulingprobleme verallgemeinert. Wir haben die vorgestellten Propagierungsfunktionen auf Monotonie untersucht und gezeigt, welche Teile der zugrundeliegenden Algorithmen für monotone Funktionen verwendet werden können. Wir haben gezeigt, wie Serialisierer durch zustandsabhängige Distribuierungsstrategien modelliert und wie durch Serialisierer redundante Projektoren zu Konfigurationen hinzugefügt werden können.

Wir haben das Modell von Propagierung in das Berechnungsmodell von Oz integriert. Dabei haben wir insbesondere aufgezeigt, welche Entscheidungen für die Implementierung der Schedulingtechniken getroffen wurden, um diese in das Berechnungsmodell von Oz zu übernehmen. Wir haben eine ausführliche Darstellung eines Implementierungsmodells für Propagierung in einer nebenläufigen Constraintsprache angegeben. Insbesondere haben wir eine Reihe von Optimierungen sowie Techniken beschrieben, um Probleme aus der Praxis effizient zu lösen.

Wir haben eine interaktive, grafische Werkbank zur Lösung von disjunktiven und kumulativen Schedulingproblemen realisiert, womit verschiedene Kapazitätsconstraints und Serialisierer zur Problemlösung kombiniert werden können. Außerdem haben wir gezeigt, wie Iterative Verbesserung mit Constraintprogrammierung in Oz effizient verbunden werden kann. Wir haben eine Evaluierung der in Oz verfügbaren Schedulingtechniken für Job-Shop-Probleme vorgenommen, die so umfassend noch nicht durchgeführt wurde und haben die Entwurfsentscheidungen bei der Integration der Schedulingtechniken in eine Constraintsprache quantitativ untersucht und begründet. Wir haben der Constraintprogrammierung das neue Anwendungsfeld der taktgesteuerten Echtzeitsysteme erschlossen. Wir haben zeigen können, daß für Job-Shop-Probleme eingeführte Schedulingtechniken sich auf taktgesteuerte Echtzeitsysteme anwenden lassen und damit selbst sehr große Probleme zu lösen sind.

## 14.2 Ausblick

Diese Arbeit kann in zwei Richtungen erweitert werden. Zum einen sind dies Ergänzungen, die in dem hier vorgestellten Rahmen möglich sind oder auch schon begonnen wurden. Zum anderen können neue Wege gegangen werden, indem das System weitere Problemlösefähigkeiten integriert.

Eine naheliegende Fortsetzung unserer Arbeiten besteht darin, stärkere Propagierungsalgorithmen für vorhandene globale Constraints zu entwickeln oder neue globale Constraints zu integrieren. So können stärkere Schedulingtechniken (aber auch Distribuierungsstrategien) für kumulative Schedulingprobleme die vorhandenen Algorithmen verbessern. Noch nicht näher untersucht worden sind Propagierungsalgorithmen für den Constraint, der besagt, daß eine Menge von Rechtecken sich nicht überlappen darf (siehe Abschnitt 4.3). Dieser Constraint hat Anwendungen im Layout aber auch in verschiedenen anderen Gebieten wie Scheduling [Sim95] und würde Oz neue Anwendungen zugänglich machen.

Eine vielversprechende Technik, die bereits in Oz vorhanden ist, sind programmgesteuerte Propagierer (siehe Abschnitt 11.7.5). Gute Erfahrungen haben wir mit dieser Technik bei der effizienten Implementierung von Serialisierern gemacht. Anwendungsmöglichkeiten für programmgesteuerte Propagierer können Probleme sein, bei denen Constraints dynamisch erzeugt werden,

aus Effizienzgründen aber globale Constraints zum Einsatz kommen müssen. Hier kann dann ein einzelner globaler Constraint nach und nach über immer mehr Argumenten rechnen. Ein weiteres Anwendungsgebiet könnte das Debugging von Constraintprogrammen sein, da ein programmgesteuerter Propagierer Information über seine Argumente und seinen Zustand an seine Umwelt geben kann, aber auch sein Verhalten direkt kontrollierbar ist.

Die mit Oz gesammelte Erfahrung kann auch dazu verwendet werden, weitere Constraintsysteme zu unterstützen. Durch die Öffnung der abstrakten Maschine durch das Constraint Programming Interface [MW97a, MW97b] ergeben sich hier gute Möglichkeiten. Ein erster Schritt in diese Richtung ist die Integration von Mengen-Constraints über natürlichen Zahlen [MM97].

Über den bestehenden Rahmen hinaus geht die Kopplung von constraintbasiertem Problemlösen mit anderen Verfahren. In dieser Arbeit haben wir uns auf constraintbasierte Methoden konzentriert. Wir haben hier einige Spezialverfahren für Scheduling aus dem Operations Research integriert, doch dies konnte im Rahmen der gewählten Modellierung geschehen. Andere erfolgreiche Ansätze zur Lösung kombinatorischer Probleme haben wir bewußt ausgeblendet. Ein solches Verfahren ist zum Beispiel die Ganzzahlige Lineare Programmierung [NW88]. In jüngerer Zeit werden oft auch heuristische Verfahren wie Simulated Annealing, Tabu-Suche oder Genetische Algorithmen verwendet (einen Überblick bietet [AL97, Ree93, GL97]).

Diese anderen Verfahren müssen nicht im Gegensatz zu moderner Constraintprogrammierung stehen. Ganz im Gegenteil ist bereits eine zunehmende Integration und Kombination unterschiedlicher Techniken zu beobachten. Beispiele zur Kombination von Ganzzahliger Linearer Programmierung mit Constraintprogrammierung sind [RWH96] oder [BK97]. Verbindungen von heuristischen Verfahren mit Constraintprogrammierung findet man zum Beispiel in [PGR97, MVH97] oder in Kapitel 9. Gerade Oz als eine so expressive Programmiersprache sollte hier vielversprechende Möglichkeiten für eine Kopplung verschiedener Ansätze bieten.

# Literaturverzeichnis


[AB93] AGGOUN, A. und N. BELDICEANU: *Extending CHIP in order to solve complex scheduling and placement problems*. Journal of Mathematical and Computer Modelling, 17(7):57–73, 1993.

[ABZ88] ADAMS, J., E. BALAS und D. ZAWACK: *The shifting bottleneck procedure for job shop scheduling*. Management Science, 34(3):391–401, 1988.

[AC91] APPLEGATE, D. und W. COOK: *A computational study of the job-shop scheduling problem*. Operations Research Society of America, Journal on Computing, 3(2):149–156, 1991.

[AL97] AARTS, E. und J.K. LENSTRA: *Local Search in Combinatorial Optimization*. John Wiley, 1997.

[Bak74] BAKER, K. R.: *Introduction to Sequencing and Scheduling*. Wiley & Sons, 1974.

[Bap94] BAPTISTE, P.: *Constraints-based scheduling: two extensions*. Masters's thesis, University of Strathclyde, 1994.

[Bar83] BARTUSCH, M.: *Optimierung von Netzplänen mit Anordnungsbeziehungen bei knappen Betriebsmitteln*. Doktorarbeit, Univ. Passau, Fakultät für Mathematik und Informatik, 1983.

[BC94] BELDICEANU, N. und E. CONTEJEAN: *Introducing global constraints in CHIP*. Journal of Mathematical and Computer Modelling, 20(12):97–123, 1994.

[BC97] BLEUZEN GUERNALEC, N. und A. COLMERAUER: *Narrowing a* $2n$*-block of sortings in* $O(n \log n)$. In: SMOLKA, G. (Herausgeber): *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Band 1330 der Reihe *Lecture Notes in Computer Science*, Seiten 2–16. Springer-Verlag, 1997.

[BDP96] BOUIZUMAULT, P., Y. DELON und L. PERIDY: *Constraint logic programming for examination timetabling*. Journal of Logic Programming, 5:345–374, 1996.

[BDSF97a] BECK, J.C., A.J. DAVENPORT, E.M. SITARSKI und M.S. FOX: *Beyond contention: Extending texture-based scheduling heuristics*. In: *Proceedings of the AAAI National Conference on Artificial Intelligence*, 1997.

[BDSF97b] BECK, J.C., A.J. DAVENPORT, E.M. SITARSKI und M.S. FOX: *Texture-based heuristics for scheduling revisited*. In: *Proceedings of the AAAI National Conference on Artificial Intelligence*, 1997.

[Ben96] BENHAMOU, F.: *Heterogenous constraint solving*. In: *Algebraic and Logic Programming*, Band 1139 der Reihe *Lecture Notes in Computer Science*, Seiten 62–76. Springer-Verlag, 1996.

[BESW94] BŁAZEWICZ, J., K.H. ECKER, G. SCHMIDT und J. WĘGLARZ: *Scheduling in Computer and Manufacturing Systems*. Springer Verlag, 2. Auflage, 1994.

[BFR95] BESSIERE, C., E. FREUDER und J.-C. REGIN: *Using inference to reduce arc consistency computation*. In: *Proceedings of the International Joint Conference on Artificial Intelligence*, Seiten 592–598, 1995.

[BG96] BENHAMOU, F. und L. GRAINVILLIERS: *Combining local consistency, symbolic rewriting and interval methods*. In: *Proccedings of AISMC-3*, Nummer 1138 in *Lecture Notes in Computer Science*, Seiten 144–159. Springer-Verlag, 1996.

[BK97] BOCKMAYR, A. und T. KASPER: *A unifying framework for integer and finite domain constraint programming*. Research Report MPI-I-97-2-008, Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany, 1997.

[BKST97] BRUCKER, P., S. KNUST, A. SCHOO und O. THIELE: *A branch & bound algorithm for the resource-constrained project scheduling problem*. Osnabrücker Schriften zur Mathematik, Reihe P, No. 192, Universität Osnabrück, 1997.

[BLK83] BŁAZEWICZ, J., J.K. LENSTRA und A.H.G. RINNOOY KAN: *Scheduling subject to resource constraints: Classification and complexity*. Discrete Applied Mathematics, 5:11–24, 1983.

[BLP97] BAPTISTE, P. und C. LE PAPE: *Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems*. In: SMOLKA, G. (Herausgeber): *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Band 1330 der Reihe *Lecture Notes in Computer Science*, Seiten 376–390. Springer-Verlag, 1997.

[BPN95a] BAPTISTE, P., C. LE PAPE und W. NUIJTEN: *Constraint-based optimization and approximation for job-shop scheduling*. In: *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI-95*, 1995.

[BPN95b] BAPTISTE, P., C. LE PAPE und W. NUIJTEN: *Incorporating efficient Operations Research algorithms in constraint-based scheduling*. In: *First International Joint Workshop on Artificial Intelligence and Operations Research*, 1995.

[BR65] BERTIER, P. und B. ROY: *Trois examples numeriques d'application de la procedure SEP*. Note de travail No. 32 de la Direction Scientifique de la SEMA, 1965.

[BR75] BITNER, J. und E.M. REINGOLD: *Backtrack programming techniques*. Communications of the ACM, 18(11):651–655, 1975.

[BR96] BURKE, E.K. und P. ROSS (Herausgeber): *Practice and Theory of Automated Timetabling, First International Conference, Selected Papers, Edinburgh 1995*, Band 1153 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[CA95] CHENG, S.T. und A.K. AGRAWALA: *Allocation and scheduling of real-time periodic tasks with realative timing constraints*. In: *Second International Workshop on Real-Time Computing Systems and Applications*, 1995.

[Car87] CARLSSON, M.: *Freeze, indexing, and other implementation issues in the WAM*. In: *Proceedings of 4th International Conference on Logic Programming*, Seiten 40–58. The MIT Press, May 1987.

[Car95] CARLSON, B.: *Compiling and executing Finite Domain constraints*. Uppsala Theses in Computing Science No. 21, Uppsala University, Box 311, S-751 05 Uppsala, Sweden, 1995.

[CC95] CARLSON, B. und M. CARLSSON: *Compiling and Executing Disjuctions of finite domain constraints*. In: *Proceedings of the International Conference on Logic Programming*, Seiten 117–131. The MIT Press, 1995.

[CCD94] CARLSON, B., M. CARLSSON und D. DIAZ: *Entailment of finite domain constraints*. In: HENTENRYCK, P. VAN (Herausgeber): *Proceedings of the International Conference on Logic Programming*, Seiten 339–353. The MIT Press, 1994.

[CCJ95] CARLSON, B., M. CARLSSON und S. JANSON: *The implementation of AKL(FD)*. In: *Proceedings of the International Symposium on Logic Programming*, Seiten 227–241. The MIT Press, 1995.

[CD93] CODOGNET, P. und D. DIAZ: *Boolean Constraint Solving Using clp(FD)*. In: *Proceedings of the International Symposium on Logic Programming*. MIT Press, 1993.

[CD95] CODOGNET, P. und D. DIAZ: `wamcc`*: Compiling Prolog to C*. In: STERLING, L. (Herausgeber): *Proceedings of the International Conference on Logic Programming*, Seiten 317–322, Kanagawa, Japan, 1995. The MIT Press.

[CD96] CODOGNET, P. und D. DIAZ: *Compiling constraints in* `clp(FD)`. Journal of Logic Programming, 27(3):185–226, 1996.

[CF95] CHAMARD, A. und A. FISCHLER: *CHIC lesons on CLP methodology*. In: *Practical Application of Constraint Technology*, Seiten 67–71, 1995. A more detailed version is available through http://www.ecrc.de/eclipse/html/CHIC_Methodology.html.

[CL94a] CASEAU, Y. und F. LABURTHE: *Improved CLP scheduling with task intervals*. In: *Proceedings of the International Conference on Logic Programming*, Seiten 369–383. The MIT Press, 1994.

[CL94b] CASEAU, Y. und F. LABURTHE: *Introduction to the CLAIRE programming language, Version 1.040*. Laboratoire Mathematiques et Informatique de l'Ecole Normale Superieure, 1994.

[CL95] CASEAU, Y. und F. LABURTHE: *Disjunctive scheduling with task intervals*. LIENS Technical Report 95-25, Laboratoire d'Informatique de l'Ecole Normale Superieure, 1995.

[CL96a] CASEAU, Y. und F. LABURTHE: *CLAIRE code for job-shop scheduling*. `http://www.dmi.ens.fr/users/laburthe/jobshop/index.html`, 1996.

[CL96b] CASEAU, Y. und F. LABURTHE: *Cumulative scheduling with task intervals*. In: *Joint International Conference and Symposium on Logic Programming*, Seiten 363–377. The MIT Press, 1996.

[CL96c] CASEAU, Y. und F. LABURTHE: *Solving small TSPs with constraints*. In: *Proceedings of the International Conference on Logic Programming*, Seiten 316–330. The MIT Press, 1996.

[CL97a] CASEAU, Y. und F. LABURTHE: *A constrained based approach to the RCPSP*. In: DAVENPORT, A. und C. BECK (Herausgeber): *CP97 Workshop on Industrial Constraint-Directed Scheduling*, `http://www.ie.utoronto.ca/EIL/profiles/andrewd/schedulingWS.html`, 1997.

[CL97b] CASEAU, Y. und F. LABURTHE: *Solving various weighted matching problems with constraints*. In: SMOLKA, G. (Herausgeber): *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Nummer 1330 in *Lecture Notes in Computer Science*, Seiten 17–31. Springer-Verlag, 1997.

[COC97] CARLSSON, M., G. OTTOSON und B. CARLSON: *An open-ended finite domain constraint solver*. In: *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, Band 1292 der Reihe *Lecture Notes in Computer Science*, Seiten 191–206. Springer-Verlag, 1997.

[Col90] COLMERAUER, A.: *An Introduction to PROLOG III*. Communications of the ACM, Seiten 70–90, Juli 1990.

[Col96] COLOMBANI, Y.: *Constraint programming: an efficient and practical approach to solving the job-shop problem*. In: *Principles and Practice of Constraint Programming*, Band 1118 der Reihe *Lecture Notes in Computer Science*, Seiten 149–163. Springer-Verlag, 1996.

[COS96] COSYTEC, Orsay, France: *CHIP, Version 5.0.1, Reference Manual*, 1996.

[CP89] CARLIER, J. und E. PINSON: *An algorithm for solving the job-shop problem*. Management Science, 35(2):164–176, 1989.

[CP90] CARLIER, J. und E. PINSON: *A practical use of Jacksons's preemptive schedule for solving the job shop problem*. Annals of Operations Research, 26:269–287, 1990.

[CP94] CARLIER, J. und E. PINSON: *Adjustment of heads and tails for the job-shop problem*. European Journal of Operational Research, 78:146–161, 1994.

[CS94] CHENG, C.-C. und S.F. SMITH: *Generating feasible schedules under complex metric constraints*. In: *Proceedings of the AAAI National Conference on Artificial Intelligence*, Seiten 1086–1091, 1994.

[DC93] DIAZ, D. und P. CODOGNET: *A minimal extension of the WAM for clp(FD)*. In: *Proceedings of the International Conference on Logic Programming*, Seiten 774–790, Budapest, Hungary, 1993. MIT Press.

[Dec92] DECHTER, R.: *From local to global consistency*. Artificial Intelligence, 55:87–107, 1992.

[Dem92] DEMEULEMEESTER, E.: *Optimal algorithms for various classes of multiple resource constrained project scheduling problems*. Doktorarbeit, Katholieke Universiteit Leuven, Belgium, 1992.

[DH92] DEMEULEMEESTER, E. und W. HERROELEN: *A branch-and-bound procedure for the multiple resource-constrained project scheduling problem*. Management Science, 38(12):1803–1818, 1992.

[DSV88a] DINCBAS, M., H. SIMONIS und P. VAN HENTENRYCK: *Solving a Cutting-Stock Problem in CLP*. In: *Proceedings of the International Conference on Logic Programming*, Seiten 42–58. The MIT Press, 1988.

[DSV88b] DINCBAS, M., H. SIMONIS und P. VAN HENTENRYCK: *Solving the car sequencing problem in constraint logic programming*. In: *European Conference on Artificial Intelligence*, Seiten 290–295, 1988.

[DSV90] DINCBAS, M., H. SIMONIS und P. VAN HENTENRYCK: *Solving Large Combinatorial Problems in Logic Programming*. Journal of Logic Programming, 8:75–93, 1990.

[DT93] DELL'AMICO, M. und M. TRUBIAN: *Applying tabu search to the job-shop scheduling problem*. Annals of Operations Research, 41:231–252, 1993.

[DVS+88] DINCBAS, M., P. VAN HENTENRYCK, H. SIMONIS, A. AGGOUN, T. GRAF und F. BERTHIER: *The Constraint Logic Programming Language CHIP*. In: *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, Seiten 693–702, Tokyo, Japan, Dezember 1988.

[ECR96] ECRC: *ECL$^i$PS$^e$, User Manual Version 3.5.2*, December 1996.

[ELT91] ERSCHLER, J., P. LOPEZ und C. THURIOT: *Raisonnement temporel sous contraintes de ressources et problèmes d'ordonnancement*. Revue d'Intellicence Artificielle, 5(3):7–32, 1991.

[Eri97] ERIKSSON, C.: *A framework for the design of distributed real-time systems*. ISRN KTH/MMK/R–97/2-SE, Royal Institute of Technology, KTH, Stockholm, Sweden, 1997.

[ERV76] ERSCHLER, J., F. ROUBELLAT und J.P. VERNHES: *Finding some essential characteristics of the feasible solutions for a scheduling problem*. Operations Research, 24:772–782, 1976.

[ERV80] ERSCHLER, J., F. ROUBELLAT und J.P. VERNHES: *Characterizing the set of feasible sequences for n jobs to be carried out on a single machine*. European Journal of Operations Research, 4:189–194, 1980.

[Fox94] FOX, M.S.: *ISIS: A retrospective*. In: ZWEBEN, M. und M.S. FOX (Herausgeber): *Intelligent Scheduling*, Seiten 3–28. Morgan Kaufmann Publishers, 1994.

[Fre82] FRENCH, S.: *Sequencing and scheduling: An introduction to the mathematics of the job-shop*. Wiley, 1982.

[Fre88] FREUDER, E.: *Backtrack-free and backtrack-bounded search*. In: KANAL, L. und V. KUMAR (Herausgeber): *Search in Artifical Intelligence*, Seiten 343–369. Springer-Verlag, 1988.

[Frü95] FRÜHWIRTH, T.: *Constraint handling rules*. In: PODELSKI, A. (Herausgeber): *Constraint Programming: Basics and Trends*, Band 910 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[FS84] FOX, M.S. und S.F. SMITH: *ISIS: A knowledge-based system for factory scheduling*. Expert Systems, 1(1):25–49, 1984.

[FS86] FALKOWSKI, B.-J. und L. SCHMITZ: *A Note on the Queens' Problem*. Information Processing Letters, 23:39–46, Juli 1986.

[FS95] FROMHERZ, M. und V. SARASWAT: *Model-based computing: using concurrent constraint programming for modeling and model compilation*. In: *Principles and Practice of Constraint Programming*, Band 976 der Reihe *Lecture Notes in Computer Science*, Seiten 629–635. Springer-Verlag, 1995.

[GH88] GÜSGEN, H.-W. und J. HERTZBERG: *Some fundamental properties of local constraint propagation*. Artificial Intelligence, 36:237–247, 1988.

[GJ79] GAREY, M.R. und D.S. JOHNSON: *Computers and Intractability*. W.H. Freeman and company, 1979.

[GL97] GLOVER, F. und M. LAGUNA: *Tabu Search*. Kluwer Academic Publishers, 1997.

[HDdR96] HERROELEN, W., E. DEMEULEMEESTER und B. DE REYCK: *Resource-constrained project scheduling*. Technical Report No. 9644, Katholieke Universiteit Leuven, 1996.

[HE80] HARALICK, R.M. und G.L. ELLIOTT: *Increasing tree search efficiency for constraint satisfaction problems*. Artificial Intelligence, 14:263–313, 1980.

[Hen97a] HENZ, M.: *Objects for Concurrent Constraint Programming*. Kluwer Academic Publishers, 1997.

[Hen97b] HENZ, M.: *The Oz Notation*. DFKI Oz Documentation Series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1997.

[HG95] HARVEY, W.D. und M.L. GINSBERG: *Limited discrepancy search*. In: *Proceedings of the International Joint Conference on Artificial Intelligence*, Seiten 607–613, 1995.

[HJ90] HARIDI, S. und S. JANSON: *Kernel Andorra Prolog and its Computation Model*. In: WARREN, D.H.D. und P. SZEREDI (Herausgeber): *Logic Programming, Proceedings of the 7th International Conference*, Seiten 31–48, Cambridge, MA, June 1990. The MIT Press.

[HMSW97] HENZ, M., M. MÜLLER, C. SCHULTE und J. WÜRTZ: *The Oz Standard Modules*. DFKI Oz Documentation Series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1997.

[Hol90] HOLZBAUR, C.: *Specification of constraint based inference mechanisms through extended unification*. Doktorarbeit, Dept. of Medical Cybernetics and AI, Technische Universität Wien, 1990.

[HSW93] HENZ, M., G. SMOLKA und J. WÜRTZ: *Oz - A Programming language for multi-agent systems*. In: *13th International Joint Conference on Artificial Intelligence*, Seiten 404–409, Chambéry, France, 1993. Morgan Kaufmann Publishers.

[HSW95] HENZ, M., G. SMOLKA und J. WÜRTZ: *Object-oriented Concurrent Constraint Programming in Oz*. In: SARASWAT, V. und P. VAN HENTENRYCK (Herausgeber): *Principles and Practice of Constraint Programming*, Kapitel 2, Seiten 27–48. The MIT Press, Cambridge, MA, 1995.

[Hue80] HUET, G.: *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*. Journal of the ACM, 27(4):797–821, 1980.

[HW96a] HENZ, M. und J. WÜRTZ: *Constraint-Based Time Tabling – A Case Study*. Journal of Applied Artificial Intelligence, 10(5):439–453, October 1996.

[HW96b] HENZ, M. und J. WÜRTZ: *Using Oz for College TimeTabling*. In: BURKE, E.K. und P. ROSS (Herausgeber): *Practice and Theory of Automated Timetabling, First International Conference, Selected Papers, Edinburgh 1995*, Band 1153 der Reihe *Lecture Notes in Computer Science*, Seiten 162–178. Springer-Verlag, 1996.

[ILOG96a] ILOG, URL: `http://www.ilog.com`: ILOG SCHEDULER *2.2, User Manual, 1996*, 1996.

[ILOG96b] ILOG, URL: `http://www.ilog.com`: ILOG SOLVER *3.2, User Manual, 1996*, 1996.

[Int95] INTELLIGENT SYSTEMS LABORATORY: *SICStus Prolog User's Manual*. SICS Research Report, Swedish Institute of Computer Science, URL http://www.sics.se/isl/sicstus.html, 1995.

[Jac56] JACKSON, J. R.: *An Extension of Johnson's Result on Job Lot Scheduling*. Naval Research Logistics Quarterly, 3:201–203, 1956.

[Jan94] JANSON, S.: *AKL – A multiparadigm programming language*. Doktorarbeit, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, 1994.

[JL87] JAFFAR, J. und J.-L. LASSEZ: *Constraint logic programming*. In: *Proceedings of the ACM Symposium on Principles of Programming Languages*, Seiten 111–119, 1987.

[JM94] JAFFAR, J. und M. MAHER: *Constraint Logic programming - A Survey*. Journal of Logic Programming, 19/20:503–582, 1994.

[JMSY92] JAFFAR, J., S. MICHAYOV, P. J. STUCKEY und R. H. C. YAP: *The CLP($\mathcal{R}$) language and system*. ACM Transactions on Programming Languages and Systems, 14(3):339–395, 1992.

[JS93] JOURDAN, J. und T. SOLA: *The versatility of handling disjunctions as constraints*. In: *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, Seiten 60–74. Springer-Verlag, 1993.

[Kan76] KAN, A.H.G. RINNOOY: *Machine scheduling problems: classification, complexity and computations*. Nijhoff, The Hague, 1976.

[Kop91] KOPETZ, H.: *Event-Triggered versus time-triggered real-time systems*. In: *Proceedings of International Workshop on Operating Systems of the 90s and Beyond*, Band 563 der Reihe *Lecture Notes in Computer Science*, Seiten 87–101. Springer-Verlag, Berlin, Germany, 1991.

[Kow74] KOWALSKI, R.: *Predicate logic as a programming language*. In: ROSENFELD, J. (Herausgeber): *Proceedings of the World Computer Congress of the IFIP*, Seiten 569–574. North-Holland, 1974.

[Kum92] KUMAR, V.: *Algorithms for constraint-satisfaction problems: A survey*. AI Magazine, Spring:32–44, 1992.

[Law84] LAWRENCE, S.: *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques*. Technischer Bericht GSIA, Carnegie Mellon University, 1984.

[LL73] LIU, C.L. und J.W. LAYLAND: *Scheduling algorithms for multiprogramming in a hard real-time environment*. Journal of the ACM, 20(1):46–61, 1973.

[Loc96] LOCK, H.C.R.: *Optimizing the cumulative constraint by global capacity constraints*. In: *Workshop Planen und Konfigurieren*, Seiten 137–148. infix Verlag, 1996.

[lPW93] PROVOST, T. LE und M. WALLACE: *Generalized constraint propagation over the CLP scheme*. Journal of Logic Programming, 16:319–359, 1993.

[Mac77] MACKWORTH, A. K.: *Consistency in Networks of Relations*. Artificial Intelligence, 8:99–118, 1977.

[MAC+89] MEIER, M., A. AGGOUN, D. CHAN, P. DUFRESNE, R. ENDERS, D.H. DE VILLENEUVE, A. HEROLD, P. KAY, B. PEREZ, E. VAN ROSSUM und J. SCHIMPF: *SEPIA - an extendible Prolog system*. In: *11th World Computer Congress IFIP'89*, Seiten 1127–1132, 1989.

[Mah87] MAHER, M.J.: *Logic semantics for a class of committed-choice programs*. In: LASSEZ, J.-L. (Herausgeber): *Logic Programming, Proceedings of the Fourth International Conference*, Seiten 858–876, 1987.

[Meh98] MEHL, M.: *An Abstract Machine for Oz – Records, Threads, Deep Guards*. Doktorarbeit, Universität des Saarlandes, Fachbereich Informatik, Postfach 1150, D-66041 Saarbrücken, Germany, 1998. In Vorbereitung.

[MF85] MACKWORTH, A. K. und E. C. FREUDER: *The complexity of some polynomial network consistency algorithms for constraint satisfaction problems*. Artificial Intelligence, 25:65–74, 1985.

[Min96] MINTON, S.: *Automatically configuring constraint satisfaction programs: a case study*. Constraints, 1(1&2):7–43, 1996.

[MJM96] M. JAMPEL, E. FREUDER und M. MAHER (Herausgeber): *Over-Constrained Systems*, Band 1106 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[MM97] MÜLLER, T. und M. MÜLLER: *Finite Set Constraints in Oz*. In: *13. Workshop Logische Programmierung*, Technische Universität München, 1997.

[MMP+97] MEHL, M., T. MÜLLER, K. POPOV, R. SCHEIDHAUER und C. SCHULTE: *DFKI Oz User's Manual*. DFKI Oz Documentation Series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1997.

[Mon74] MONTANARI, U.: *Networks of constraints: fundamental properties and application to picture processing*. Information Sciences, 7:95–132, 1974. Also appeared as Technical Report, Carnegie Mellon University, 1970.

[MS96] MARTIN, P. und D.B. SHMOYS: *A new approach to computing optimal schedules for the job shop scheduling problem*. In: *International Conference on Integer Programming and Combinatorial Optimization, Vancouver*, Seiten 389–403, 1996.

[MS97] MEHL, M. und C. SCHULTE: *Window programming in DFKI Oz*. DFKI Oz Documentation Series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1997.

[MSS95] MEHL, M., R. SCHEIDHAUER und C. SCHULTE: *An Abstract Machine for Oz*. In: HERMENEGILDO, M. und S. D. SWIERSTRA (Herausgeber): *Programming Languages: Implementations, Logics and Programs, 7th International Symposium,*

*PLILP'95*, Band 982 der Reihe *Lecture Notes in Computer Science*, Seiten 151–168. Springer-Verlag, 1995.

[MT63] MUTH, J.F. und G.L. THOMPSON: *Industrial Scheduling*. Prentice Hall, 1963.

[Mül95] MÜLLER, T.: *Adding Constraint Systems to DFKI Oz*. In: *WOz'95, International Workshop on Oz Programming*, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, 29 November–1 December 1995.

[MVH97] MICHEL, L. und P. VAN HENTENRYCK: LOCALIZER*: a modeling language for local search*. In: SMOLKA, G. (Herausgeber): *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Band 1330, Seiten 238–252, 1997.

[MW95] MÜLLER, T. und J. WÜRTZ: *Constructive Disjunction in Oz*. In: *11th Workshop Logic Programming*, Seiten 113–122, Vienna, Austria, 1995. GMD-Studien Nr. 270, ISBN 3-88457-270-9.

[MW96] MÜLLER, T. und J. WÜRTZ: *Interfacing Propagators with a Concurrent Constraint Language*. In: *JICSLP96 Post-conference workshop and Compulog Net Meeting on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, Seiten 195–206, Bonn, Germany, 1996.

[MW97a] MÜLLER, T. und J. WÜRTZ: *The constraint propagator interface of DFKI Oz*. DFKI Oz Documentation Series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1997.

[MW97b] MÜLLER, T. und J. WÜRTZ: *Extending a Concurrent Constraint Language by Propagators*. In: MALUSZYNSKI, J. (Herausgeber): *International Logic Programming Symposium*, Seiten 149–163. The MIT Press, 1997.

[NA96] NUIJTEN, W.P.M. und E.H.L. AARTS: *A computational study of constraints satisfaction for multiple capacitated job shop scheduling*. European Journal of Operational Research, 1996.

[Nad88] NADEL, B.: *Tree search and arc consistency in constraint-satisfaction algorithms*. In: KANAL, L. und V. KUMAR (Herausgeber): *Search in Artificial Intelligence*, Seiten 287–342. Springer-Verlag, 1988.

[NS93] NOWICKI, E. und C. SMUTNICKI: *A fast taboo search algorithm for the job-shop problem*. Preprint 8/93, Institute of Engineering Cybernetics, Technical University of Wroclaw, 1993.

[Nui94] NUIJTEN, W.P.M.: *Time and resource constrained scheduling*. Doktorarbeit, Technical University Eindhoven, 1994.

[NW88] NEMHAUSER, G. L. und L. A. WOLSEY: *Integer and combinatorial optimization*. John Wiley and Sons, 1988.

[OB93] OLDER, W. und F. BENHAMOU: *Programming in CLP(BNR)*. In: *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, Seiten 239–249, Newport, RI, USA, 1993.

[PB96] PAPE, C. LE und P. BAPTISTE: *Constraint propagation technqiues for disjunctive scheduling: the preemptive case*. In: *European Conference on Artificial Intelligence*, Seiten 619–623, 1996.

[PB97] PAPE, C. LE und P. BAPTISTE: *A constraint programming library for preemptive and non-preemptive scheduling*. In: *Practical Application of Constraint Technology*, Seiten 237–256. The Practical Application Company, 1997.

[PGR97] PESANT, G., M. GENDREAU und J.-M. ROUSSEAU: *GENIUS-CP: a generic single-vehicle routing algorithm*. In: SMOLKA, G. (Herausgeber): *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Band 1330, Seiten 420–334. Springer-Verlag, 1997.

[PL95] PUGET, J.-F. und M. LECONTE: *Beyond the glass box: constraints as objects*. In: *Proceedings of the International Symposium on Logic Programming*, Seiten 513–527. The MIT Press, 1995.

[Pro97] PROGRAMMING SYSTEMS LAB: *The Oz Programming System*. Universität des Saarlandes, `http://www.ps.uni-sb.de/www/oz/`, 1997.

[Pug94] PUGET, J.-F.: *A C++ implementation of CLP*. In: *Ilog Solver collected papers, Ilog technical report*, 1994.

[Ree93] REEVES, C. R.: *Modern heuristic techniques for combinatorial problems*. Halsted Press, 1993.

[Reg94] REGIN, J.-C.: *A filtering algorithm for constraints of difference in CSPs*. In: *Proceedings of the AAAI National Conference on Artificial Intelligence*, Seiten 362–367, 1994.

[RP97] REGIN, J.-C. und J.-F. PUGET: *A filtering algorithm for global sequencing constraints*. In: SMOLKA, G. (Herausgeber): *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Band 1330 der Reihe *Lecture Notes in Computer Science*, Seiten 32–46. Springer-Verlag, 1997.

[RWH96] RODOSEK, R., M.G. WALLACE und M.T. HAIJAN: *A new approach to integrate mixed integer programming with CLP*. In: *Proceedings of the Workshop on Constraint Programming Applications, in conjunction with the Second International Conference on Principles and Practice of Constraint Programming (CP96)*, 1996.

[Sad91] SADEH, N.: *Look-ahead techniques for micro-opportunistic job shop scheduling*. Research Report CMU-CS-91-102, School of Computer Science, Carnegie Mellon University, 1991.

[Sad94] SADEH, N.: *The Micro-Boss factory scheduler*. In: ZWEBEN, M. und M.S. FOX (Herausgeber): *Intelligent Scheduling*, Seiten 99–136. Morgan Kaufmann Publishers, 1994.

[Sar93] SARASWAT, V.: *Concurrent constraint programming*. MIT Press, Cambridge, MA, 1993.

[SC93] SMITH, S.F. und C.-C. CHENG: *Slack-based heuristics for constraint satisfaction scheduling*. In: *Proceedings of the AAAI National Conference on Artificial Intelligence*, Seiten 139–144, 1993.

[Sch95] SCHAERF, A.: *A Survey of Automated Timetabling*. Technischer BerichtCS-R9567, CWI, Amsterdam, NL, 1995.

[Sch97a] SCHULTE, C.: *Oz Explorer: A Visual Constraint Programming Tool*. In: NAISH, L. (Herausgeber): *Proceedings of the Fourteenth International Conference on Logic Programming*, Seiten 286–300. The MIT Press, 1997.

[Sch97b] SCHULTE, C.: *Programming Constraint Inference Engines*. In: SMOLKA, G. (Herausgeber): *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Band 1330 der Reihe *Lecture Notes in Computer Science*, Seiten 520–534. Springer-Verlag, 1997.

[Sch98a] SCHEIDHAUER, R.: *Über die effiziente Implementierung der nebenläufigen Constraint-Sprache Oz*. Doktorarbeit, Universität des Saarlandes, Fachbereich Informatik, Postfach 1150, D-66041 Saarbrücken, Germany, 1998. In Vorbereitung.

[Sch98b] SCHULTE, C.: *Constraint Inference Engines*. Doktorarbeit, Universität des Saarlandes, Fachbereich Informatik, Postfach 1150, D-66041 Saarbrücken, Germany, 1998. In Vorbereitung.

[Sha89] SHAPIRO, E.: *The family of concurrent logic programming languages*. ACM Computing Surveys, 21(3):413–510, 1989.

[Sim95] SIMONIS, H.: *Modelling machine set-up time in CHIP*. Technical Report, COSYTEC, 1995.

[Smi94] SMITH, S.F.: *OPIS: A methodology and architecture for reactive scheduling*. In: ZWEBEN, M. und M.S. FOX (Herausgeber): *Intelligent Scheduling*, Seiten 29–66. Morgan Kaufmann Publishers, 1994.

[Smo95] SMOLKA, G.: *The Oz Programming Model*. In: LEEUWEN, J. VAN (Herausgeber): *Computer Science Today*, Band 1000 der Reihe *Lecture Notes in Computer Science*, Seiten 324–343. Springer-Verlag, Berlin, 1995.

[SOP+86] SMITH, S.F., P.S. OW, C. LE PAPE, B. MCLAREN und N. MUSCETTOLA: *Integrating multiple scheduling perspectives to generate detailed production plans*. In: *SME Conference on AI in Manufactering*, Seiten 123–137, 1986.

[SR90] SARASWAT, V.A. und M. RINARD: *Concurrent Constraint Programming*. In: *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, Seiten 232–245, San Francisco, CA, January 1990.

[SS94] SCHULTE, C. und G. SMOLKA: *Encapsulated Search in Higher-order Concurrent Constraint Programming*. In: BRUYNOOGHE, M. (Herausgeber): *Logic Programming: Proceedings of the 1994 International Symposium*, Seiten 505–520, Ithaca, New York, USA, November 1994. MIT-Press.

[SSW94] SCHULTE, C., G. SMOLKA und J. WÜRTZ: *Encapsulated Search and Constraint Programming in Oz*. In: BORNING, A.H. (Herausgeber): *Principles and Practice of Constraint Programming*, Band 874 der Reihe *Lecture Notes in Computer Science*, Seiten 134–150, Orcas Island, Washington, USA, 1994. Springer-Verlag.

[SSW98] SCHULTE, C., G. SMOLKA und J. WÜRTZ: *Finite Domain Constraint Programming in Oz – A Tutorial*. DFKI Oz Documentation Series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1998.

[ST94] SMOLKA, G. und R. TREINEN: *Records for Logic Programming*. Journal of Logic Programming, 18(3):229–258, April 1994.

[SW97] SCHILD, K. und J. WÜRTZ: *Scheduling of Time-Triggered Real-Time Systems*. DFKI First Publication Archive,WWW published Paper FAI-97-001, `http://www.dfki.de/epress/fpa/97/1.html`, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, 1997.

[SW98] SCHILD, K. und J. WÜRTZ: *Off-Line Scheduling of a Real-Time System*. In: *Symposium on Applied Computing*, Seiten 29–38. ACM, 1998.

[Tsa93] TSANG, E.: *Foundations of Constraint Satisfaction*. Computation in Cognitive Science. Academic Press, 1993.

[VAL94] VAESSENS, R.J.M., E.H.L. AARTS und J.K. LENSTRA: *Job shop scheduling by local search*. Eindhoven University of Technology, Department of Mathematics and Computing Science, COSOR Memorandum 94-05, 1994.

[Van89] VAN HENTENRYCK, P.: *Constraint Satisfaction in Logic Programming*. Programming Logic Series. The MIT Press, Cambridge, MA, 1989.

[Van94] VAN HENTENRYCK, P.: *Scheduling and packing in the constraint language* `cc(FD)`. In: ZWEBEN, M. und M. S. FOX (Herausgeber): *Intelligent scheduling*, Seiten 137–167. Morgan Kaufmann Publishers, 1994.

[VD87] VAN HENTENRYCK, P. und M. DINCBAS: *Forward checking in logic programming*. In: *Proceedings of the International Conference on Logic Programming*, Seiten 229–256. MIT Press, 1987.

[VD91a] VAN HENTENRYCK, P. und Y. DEVILLE: *Operational Semantics of Constraint Logic Programming over Finite Domains*. In: *AAAI Spring Symposium Series*, Seiten 128–146. Stanford University, March 1991.

[VD91b] VAN HENTENRYCK, P. und Y. DEVILLE: *The Cardinality Operator: A New Logical Connective for Constraint Logic Programming*. In: FURUKAWA, KOICHI (Herausgeber): *Proceedings of the International Conference on Logic Programming*, Seiten 745–759, Paris, France, 1991. The MIT Press.

[VDT92] VAN HENTENRYCK, P., Y. DEVILLE und C.M. TENG: *A generic arc-consistency algorithm and its specializations*. Artificial Intelligence, 57:291–321, 1992.

[VSD91] VAN HENTENRYCK, P., V. SARASWAT und Y. DEVILLE: *Constraint Processing in cc(FD)*. Technical Report, Brown University, 1991.

[VSD92] VAN HENTENRYCK, P., H. SIMONIS und M. DINCBAS: *Constraint Satisfaction Using Constraint Logic Programming*. Artificial Intelligence, 58:113–159, 1992.

[VSD93] VAN HENTENRYCK, P., V. SARASWAT und Y. DEVILLE: *Design, Implementation and Evaluation of the Constraint Language cc(FD)*. Report CS-93-02, Brown University, January 1993. A revised version appeared as [VSD95].

[VSD95] VAN HENTENRYCK, P., V. SARASWAT und Y. DEVILLE: *Design, Implementation and Evaluation of the Constraint Language cc(FD)*. In: PODELSKI, ANDREAS (Herausgeber): *Constraints: Basics and Trends*, Band 910 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Wal75] WALTZ, D.L.: *Understanding line drawings of scenes with shadows*. In: WINSTON, P.H. (Herausgeber): *The Psychology of Computer Vision*, Seiten 19–91. McGraw Hill, 1975.

[Wal96] WALLACE, M.: *Practical applications of constraint programming*. Constraints, 1(1&2):139–168, 1996.

[WB93] WILSON, M. und A. BORNING: *Hierarchical Constraint Logic Programming*. Journal of Logic Programming, 16(3):277–318, 1993.

[WM96] WÜRTZ, J. und T. MÜLLER: *Constructive Disjunction Revisited*. In: GÖRZ, G. und S. HÖLLDOBLER (Herausgeber): *20th German Annual Conference on Artificial Intelligence*, Band 1137 der Reihe *Lecture Notes in Artificial Intelligence*, Seiten 377–386, Dresden, Germany, 1996. Springer-Verlag.

[WNS97] WALLACE, M., S. NOVELLO und J. SCHIMPF: *ECL$^i$PS$^e$: A platform for constraint logic programming*. IC-Parc, Imperial College, London, 1997.

[Wür96] WÜRTZ, J.: *Oz Scheduler: A Workbench for Scheduling Problems*. In: RADLE, M.G. (Herausgeber): *Eighth International Conference on Tools with Artificial Intelligence*, Seiten 149–156, Toulouse, France, 1996. IEEE, IEEE Computer Society Press.

[Wür97] WÜRTZ, J.: *Constraint-Based Scheduling in Oz*. In: ZIMMERMANN, U., U. DERIGS, W. GAUL, R. MÖHRIG und K.-P. SCHUSTER (Herausgeber): *Operations Research Proceedings 1996*, Seiten 218–223. Springer-Verlag, Berlin, Heidelberg, New York, 1997. Selected Papers of the Symposium on Operations Research (SOR 96), Braunschweig, Germany, September 3–6, 1996.

[Wür98] WÜRTZ, J.: *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz – Programme der Doktorarbeit*, 1998. `http://www.ps.uni-sb.de/~wuertz/Thesis/Programs`.

[ZF94] ZWEBEN, M. und M.S. FOX (Herausgeber): *Intelligent Scheduling*. Morgan Kaufman Pubishers, 1994.

[Zho97] ZHOU, J.: *A permutation-based approach for solving the job-shop problem*. Constraints, 2(2):185–213, 1997.

# Index