UNIVERSITÄT
BREMEN

Fachbereich Mathematik / Informatik
Studiengang Informatik

Diplomarbeit

YACS: Ein hybrides Framework für Constraint-Solver zur Unterstützung wissensbasierter Konfigurierung

Wolfgang Runte
Matrikelnr.: 1140989

27. Januar 2006

1. Gutachter:	Prof. Dr. Otthein Herzog Arbeitsgruppe Künstliche Intelligenz (AG KI), Universität Bremen
2. Gutachter:	Dr. Andreas Günter Hamburger Informatik Technologie-Center (HITeC), Universität Hamburg
Betreuung:	Thomas Wagner Arbeitsgruppe Künstliche Intelligenz (AG KI), Universität Bremen

Erklärung

Ich versichere, die Diplomarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Barnstorf, 27. Januar 2006

(Wolfgang Runte)

Danksagung

Mein Dank gilt Prof. Dr. Otthein Herzog für die Übernahme des Erstgutachtens und für sein Entgegenkommen bei der Gestaltung dieser Arbeit. Außerdem geht mein Dank an Dr. Andreas Günter für die Übernahme des Zweitgutachtens. Bei Thomas Wagner möchte ich mich für die ausdauernde Geduld bei der Betreuung dieser Arbeit bedanken.

Für das Korrekturlesen danke ich Pascal Bonar und Petra Burmester (trotzdem vorhandene Fehler wurden nachträglich durch den Autor eingefügt). Pascal Bonar gebührt außerdem mein Dank für seine Unterstützung bei der Installation des EngCon-Prototypen. Werner Kober und Dr. Jürgen Wessel danke ich für die vielen guten Ratschläge und die anregenden Diskussionen, die für die nötige Ablenkung abseits der Informatik gesorgt haben. Bei Mandy und Reinhard Peukert bedanke ich mich für so manches tiefschürfende und „sinnstiftende" Gespräch zu fortgeschrittener Stunde und ebenso für die spätabendliche Bewirtung nach arbeitsreichen Tagen.

Bedanken möchte ich mich auch bei allen anderen, die mich während dieser Zeit geduldig unterstützt haben, für das mir entgegengebrachte Verständnis und Vertrauen. Insbesondere danke ich meiner Familie und Petra, die mir den nötigen Rückhalt für die Bewältigung dieser Arbeit gegeben haben.

Barnstorf, im Januar 2006

Wolfgang Runte

Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

Eugene C. Freuder, Constraints, April 1997

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Definitionen

Verwendete Symbole

P	Constraint Satisfaction Problem (CSP)
C	Constraint
C^u	unäres Constraint
C^b	binäres Constraint
R	Relation
R^u	unäre Relation
R^b	binäre Relation
$R_{i,j}$	Relation zwischen den Variablen v_i und v_j
v	Constraint-Variable
D	Wertebereich einer Constraint-Variable (Domäne)
$D_1 \times \cdots \times D_n$	kartesisches Produkt der Wertebereiche D_1 bis D_n
d	Variablenwert aus der Domäne einer Constraint-Variable $d \in D$
$\vec{a}$	Teilbelegung der Variablen eines CSP
I	Intervall
$[a_1, a_2]$	abgeschlossenes Werteintervall mit a_1 als untere und a_2 als obere Schranke
$]a_1, a_2[$	offenes Werteintervall mit a_1 als untere und a_2 als obere Schranke
x^+	innerhalb der Präzisionsgrenzen die kleinste darstellbare Zahl $> x$
x^-	innerhalb der Präzisionsgrenzen die größte darstellbare Zahl $< x$
$\mathbb{R}$	Menge der reellen Zahlen
$\mathsf{I}(\mathbb{R})$	Menge der beschränkten, abgeschlossenen, reellen Intervalle (ohne $-\infty$ und $+\infty$)
$\mathsf{I}^*(\mathbb{R})$	Menge der unbeschränkten, abgeschlossenen, reellen Intervalle (inkl. $-\infty$ und $+\infty$)
$\mathsf{L}^*(\mathbb{R})$	Menge der reellwertigen, unteren Grenzen (inkl. $-\infty$ und $+\infty$)
$\mathsf{U}^*(\mathbb{R})$	Menge der reellwertigen, oberen Grenzen (inkl. $-\infty$ und $+\infty$)
$B^\square$	Box
$\Phi_{kB}(P)$	P ist kB-konsistent
$\Phi_{Box}(P)$	P ist Box-konsistent

Kapitel 1

Einleitung

Zu jedem komplexen Problem gibt es eine verblüffend triviale Lösung,
die sich einfach erklären lässt und vollkommen falsch ist.

ANONYM

An dieser Stelle wird neben einer Motivation zum Thema ein Überblick über die zu erreichenden Ziele sowie eine Übersicht über den Aufbau der vorliegenden Arbeit gegeben.

1.1 Motivation

Produktkonfigurierung ist ein stetig wachsender Bereich, dem insbesondere in Bezug auf variantenreiche Produkte, bspw. für die Angebotserstellung, eine große Bedeutung beigemessen wird. Mit unterschiedlichen, wissensbasierten Methoden lassen sich innerhalb von Konfigurierungswerkzeugen aus einzelnen Komponenten komplexe Aggregate erstellen. *Constraints*[1] sind ein Mittel zur Repräsentation von Abhängigkeiten zwischen den Komponenten einer Konfiguration. Zur Verarbeitung und Auflösung werden Constraint-Solver eingesetzt, welche die Erfüllung der Abhängigkeiten sicherstellen oder ggf. auf Inkonsistenzen innerhalb der Konfiguration aufmerksam machen (vgl. Günter und Kühn 1999; Sabin und Weigel 1998; Stumptner 1997).

Das wissensbasierte Konfigurierungswerkzeug EngCon verwendet u. a. Funktions- und Prädikat-Constraints zur Beschreibung von Abhängigkeiten zwischen Konzepten der Wissensbasis (vgl. Hollmann et al. 2000; Ranze et al. 2002). Die Auflösung der Abhängigkeiten wird von einem in EngCon integrierten Constraint-System verwaltet (vgl. Syska und Cunis 1991). Der eingesetzte Constraint-Solver wird allerdings nicht von EngCon implementiert, sondern ist derzeit eine von einem Fremdhersteller eingebundene, externe

[1] *constraint* (engl.): Einschränkung, Beschränkung, Restriktion, Randbedingung

Komponente.[2] Er ist ausschließlich für die Propagation und Auflösung von Constraints mit reellwertigen Intervalldomänen geeignet. Zudem ist die Anbindung an EngCon auf einen einzigen Constraint-Solver beschränkt und damit sehr unflexibel. Zur Erweiterung der Anwendungsmöglichkeiten des Konfigurierungswerkzeugs EngCon sowie zur Verbesserung der Effizienz ist ein Austausch oder eine Eigenimplementierung des Constraint-Solvers wünschenswert.

Der benötigte Constraint-Solver muss arithmetische Funktionen zur Berechnung der intensional in Form von algebraischen Ausdrücken formulierten Constraints innerhalb von EngCon bieten. Neben klassischen Constraint-Solvern zur Behandlung von finiten Domänen sind für die Constraint-Verarbeitung in EngCon Constraint-Lösungsmethoden für infinite, d. h. reellwertige Intervalldomänen erforderlich. Ein entsprechender Constraint-Solver muss eine hohe Präzision durch Intervallarithmetik aufweisen (z. B. für Anwendungen im Maschinenbau) sowie unabhängig von der Constraint-Domäne ein inkrementell anwachsendes Constraint-Netz propagieren können.

Aufgrund der hohen Anforderungen von EngCon, insbesondere in Bezug auf unterschiedliche zu verarbeitende Wertedomänen, existiert kein Constraint-Solver, der sämtliche Anforderungen vollständig erfüllt. Darüber hinaus ergeben sich Anforderungen an den Constraint-Lösungsmechanismus häufig in Abhängigkeit von der Aufgabenstellung des jeweiligen Konfigurierungsproblems in EngCon. Speziell benötigte Eigenschaften sind i. A. daher a priori nicht bekannt. Neben stabilen Constraint-Lösungsverfahren, die eine hohe Effizienz für möglichst viele Problemstellungen bieten, ist es deshalb erforderlich, problemabhängig unterschiedliche Verfahren nutzen zu können. Benötigt wird neben zusätzlichen Constraint-Lösungsmechanismen eine Komponente, an der sich flexibel unterschiedliche Constraint-Solver mit verschiedenen Eigenschaften, sowohl bezogen auf die Lösungsverfahren als auch auf die zu verarbeitenden Wertedomänen, einbinden lassen. Diese Constraint-Solver können eigenimplementiert aber auch Fremdsysteme sein. Eigene Implementierungen hätten den Vorteil der leichteren Erweiterbarkeit,[3] zudem entfällt der bei geeigneten Constraint-Solvern von Fremdherstellern ggf. hohe Integrationsaufwand der Komponenten.

1.2 Angestrebte Ergebnisse

Das Ziel dieser Arbeit besteht in dem Entwurf einer modularen und flexiblen Architektur, welche den problemlosen Austausch bzw. den domänenabhängigen Einsatz von Constraint-Lösungstechnologien ermöglicht. Daneben sollen eine Reihe geeigneter Lösungsverfahren

[2] Die Vorgänger von EngCon, die Konfigurierungswerkzeuge KonWerk und PlaKon, wurden in Common Lisp bzw. CLOS (*Common Lisp Object System*) implementiert. Hier fand ein eigens in Lisp implementiertes Modul zur Durchführung von arithmetischen Berechnungen Verwendung, welches die von Davis (1987) und Hyvönen (1992) beschriebenen Verfahren zur Propagation von Intervall-Constraints anwendet (vgl. Gülden 1993).

[3] Denkbare Erweiterungen, die allerdings nicht in dieser Arbeit behandelt werden, wären z. B. die Möglichkeit während der Laufzeit des Systems einzelne Constraints zurücknehmen zu können („Constraint-Relaxierung“) oder die Möglichkeit zur Beschreibung von „Constraint-Hierarchien“, in denen „harte“ und „weiche“ Constraints definiert werden können. Je nachdem auf welcher Stufe innerhalb der Hierarchie sich ein Constraint befindet, muss oder kann es erfüllt sein, um zu einer gültigen Lösung zu gelangen.

zur Constraint-Verarbeitung innerhalb von EngCon zur Verfügung gestellt werden, die möglichst robust sind und auch für unterschiedliche Problemstellungen eine möglichst stabile Performanz bieten.

Der hohe Integrationsaufwand und die Erfahrungen hinsichtlich der eingeschränkten Funktionalität und Skalierbarkeit, die bei der prototypischen Integration von Fremdsystemen zum Constraint-Lösen in EngCon gemacht wurden, sprechen für eine Eigenentwicklung: Der Aufwand für die komplexe Integration eines Fremdsystems gestaltet sich u. U. größer, als die Implementierung eigener Algorithmen zur Constraint-Verarbeitung. Vorausgesetzt das *Know-How* bzgl. der entsprechenden Constraint-Lösungsverfahren ist vorhanden. Ein wesentlicher Teil dieser Arbeit beschäftigt sich daher mit Verfahren für das effiziente Auflösen von Constraint-Problemen, sowohl über finite als auch über infinite Domänen. Dabei geht es weniger darum „das beste" Lösungsverfahren zu ermitteln, auch nicht für eine bestimmte Anwendung von EngCon.[4] Es wird vielmehr verdeutlicht, dass die Effizienz eines Constraint-Lösungsverfahrens abhängig von der in der Wissensbasis definierten Problemstellung ist. Es ist daher das Ziel, eine modulare und erweiterbare Umgebung zur Implementierung von Constraint-Solvern zur Verfügung zu stellen.

Aufgrund unterschiedlicher Domänen und der Vielfältigkeit der Problemstellungen gibt es kein generell anwendbares bzw. effizientes Constraint-Lösungsverfahren. Um innerhalb von EngCon möglichst viele Problemstellungen behandeln zu können, sind mehrere Constraint-Lösungsverfahren erforderlich. Der modulare Aufbau der Software-Architektur, in der diese Verfahren eingesetzt werden, ist dabei entscheidend für die Austauschbarkeit einzelner Komponenten. Auf diese Weise wird nicht nur sichergestellt, dass sich Lösungsmechanismen flexibel einsetzen und ggf. optimierte Constraint-Solver einbinden lassen, sondern es wird auch Raum für künftige Erweiterungen gelassen. Im Zuge einer möglichen produktiven Verwendung des Systems ist so eine Erweiterung um zusätzliche Constraint-Solver jederzeit möglich. Dies ist umso leichter zu bewerkstelligen, da die Constraint-Solver nach dem *Black-Box*-Prinzip gekapselt und unabhängig vom restlichen System arbeiten sollen.

Neben der Kombination mehrerer voneinander unabhängiger Constraint-Solver kann es weiterhin sinnvoll sein, dass Constraint-Solver aus Effizienzgründen bei der Lösung eines Problems miteinander kooperieren. Soll zudem die Problembeschreibung hinsichtlich der Wertebereiche der Constraint-Variablen domänenübergreifend möglich sein, so ist auch eine domänenübergreifende Kooperation von Constraint-Solvern in Bezug auf unterschiedliche Domänen erforderlich.

Für den Konfigurierungsvorgang von EngCon wird eine Komponente benötigt, die während der Konfigurierung die Zuweisungen übernimmt, welche Constraints von welchem konkreten Constraint-Solver verarbeitet werden sollen bzw. können. Hier liegt der Schwerpunkt der Arbeit: Die intelligente Anbindung unterschiedlicher Constraint-Lösungsverfahren in einer möglichst einfachen und erweiterbaren Architektur. Vor dem Hintergrund der Konfigurierung ist jeweils abzuwägen, welche Lösungsverfahren für welche Constraints eingesetzt werden. Für den Wissensingenieur soll sich dabei eine Abstraktion von den

[4] Hierfür wäre jeweils eine Analyse der spezifischen Wissensbasis und des darin enthaltenen Constraint-Netzes sowie empirische Tests mit unterschiedlichen Lösungsverfahren notwendig.

tatsächlich eingesetzten Lösungsverfahren ergeben. Das heißt der Wissensingenieur soll bei der Spezifikation der Constraints in der Wissensbasis aus einer frei definierbaren Liste ein Constraint-Lösungsverfahren bzw. eine Kombination von Lösungsverfahren auswählen können, mit der sich eine bestimmte Klasse von Constraint-Problemen effizient verarbeiten läßt.

Als praktische Umsetzung soll am Ende dieser Arbeit ein Prototyp stehen, an dem beispielhaft anhand implementierter Constraint-Lösungsverfahren die Funktionsweise des entwickelten Konzepts aufgezeigt und validiert werden kann.

1.3 Aufbau der Arbeit

Der Aufbau dieser Arbeit gliedert sich in die folgenden Abschnitte: In **Kapitel 2** erfolgt eine Einführung in die wissensbasierte Konfigurierung und ein Überblick über unterschiedliche Methoden zur Konfigurierung sowie existierende Konfigurierungswerkzeuge. In **Kapitel 3** wird ein detailierter Überblick über das strukturbasierte Konfigurierungswerkzeug EngCon gegeben. Es werden die konzeptionelle Architektur sowie die unterschiedlichen Wissensarten beschrieben. Außerdem enthält dieses Kapitel eine Übersicht und eine Diskussion zum Constraint-System sowie die Anforderungen von EngCon bzgl. der einzusetzenden Constraint-Lösungsverfahren. Das **Kapitel 4** enthält eine Einführung in den Constraint-Formalismus und Übersichten sowohl zu den Eigenschaften von Constraints als auch zu den bestehenden Konzepten zur Constraint-Verarbeitung. Außerdem beinhaltet dieses Kapitel eine ausführliche Evaluierung und Bewertung existierender Constraint-Systeme hinsichtlich ihrer Eignung für eine Integration in das Konfigurierungswerkzeug EngCon. Das **Kapitel 5** unterteilt sich in zwei Bereiche: Im ersten Teil des Kapitels werden Konsistenz- und Suchverfahren zum Propagieren und Lösen von klassischen Constraint-Problemen mit finiten Domänen behandelt. Dies beinhaltet Heuristiken zur Optimierung der Lösungssuche. Neben den Lösungsverfahren für binäre Constraint-Probleme werden außerdem Verfahren für n-stellige Probleme vorgestellt. Im zweiten Teil des Kapitels wird ein Überblick über Lösungsverfahren für intervallwertige Constraint-Probleme gegeben. Neben intervallarithmetischen Grundlagen beinhaltet dies Splitting-Techniken und Konsistenzalgorithmen, angewendet auf Intervalldomänen. In **Kapitel 6** wird das Konzept für ein hybrides Constraint-System entwickelt. Eingebettet in eine modulare und wiederverwendbare Framework-Architektur ist es strategiebasiert, wodurch sich Lösungsverfahren flexibel einsetzen und austauschen lassen. Zur Verarbeitung von hybriden Constraint-Problemen wird die Funktion eines Meta-Constraint-Solvers zum Lösen von heterogenen Constraints spezifiziert sowie mehrere Szenarien zur Umsetzung des Systems aufgezeigt. In **Kapitel 7** wird die Implementierung des YACS-Frameworks dargelegt. Es werden der Reihe nach die enthaltenen Komponenten aufgezählt und dokumentiert. Abschließend erfolgt die Beschreibung der Integration von YACS in das Konfigurierungswerkzeug EngCon. In **Kapitel 8** folgt eine Validierung der Umsetzung von YACS anhand der realisierten Funktionalität und der zuvor benannten Anforderungen. Neben synthetischen Problemstellungen wird die Integration des YACS-Frameworks in das strukturbasierte Konfigurierungswerkzeug EngCon validiert. Außerdem erfolgt eine Positionierung im

Vergleich mit weiterführenden Arbeiten. In **Kapitel 9** erfolgt eine Zusammenfassung sowie ein kurzer Ausblick hinsichtlich der Erweiterungsmöglichkeiten der Konzeption und des entwickelten Prototypen.

Neben den genannten Kapiteln existieren in dieser Arbeit eine Reihe von Anhängen: In **Anhang A** sind die ausführlichen Informationen über Konfigurierungswerkzeuge zu finden, die im Rahmen dieser Arbeit recherchiert und für Kapitel 2 zusammengefasst wurden. Der **Anhang B** enthält eine Beschreibung des vollständigen Konfigurierungszyklus von EngCon, der aus Gründen der Übersichtlichkeit an dieser Stelle platziert wurde. Beispiele für Constraint-Lösungsstrategien von YACS in XML-Syntax, einschließlich der zugehörigen DTD, sind in **Anhang C** dokumentiert. Der **Anhang D** enthält die Parser-Grammatik für algebraische Constraint-Ausdrücke, die von YACS interpretiert werden können. Die vollständige API-Dokumentation für das YACS-Framework mit einer Beschreibung der entwickelten Klassen und Methoden befindet sich in **Anhang E**. Ein Testprogramm, auf das während der Validierung in Kapitel 8 Bezug genommen wird, ist in **Anhang F** zu finden.

Für das bessere Verständnis werden gebräuchliche englischsprachige Begriffe als *terminus technicus* beibehalten und nicht ins Deutsche übersetzt. Ein Glossar, in dem wichtige Begriffe erläutert werden, befindet sich in **Anhang G**. Geschützte Warennamen und Warenzeichen sind innerhalb der Arbeit nicht gesondert kenntlich gemacht. Aus dem Fehlen eines solchen Hinweises kann demnach nicht geschlossen werden, dass es sich um einen freien Warennamen oder ein freies Warenzeichen handelt.

Teil I

Anwendungsdomäne und Problemstellung

Kapitel 2

Wissensbasierte Konfigurierung

Die Hälfte von dem, was man über KI hört, ist nicht wahr;
die andere Hälfte ist nicht möglich.

Derek Patridge

Im Folgenden wird eine Einführung in die wissensbasierte Konfigurierung und ein Überblick über unterschiedliche Methoden zur Konfigurierung sowie existierende Konfigurierungswerkzeuge gegeben.

2.1 Einordnung

Konfigurierung ist dem Gebiet der *Expertensysteme* – häufig auch wissensbasierte, Problemlösungs- oder Unterstützungssysteme genannt – innerhalb der Künstlichen Intelligenz (KI) zuzuordnen. Eine Definition von Expertensystemen kann der Arbeit von Günter (1992, S. 1) entnommen werden:

> „Expertensysteme sind *wissensbasierte* Programmsysteme. Sie werden für Aufgaben in schwach strukturierten Anwendungsbereichen eingesetzt, für deren Lösung Experten benötigt werden, und die mit ‘klassischen’ algorithmischen Lösungsverfahren nur unzureichend bearbeitet werden können. Dies beinhaltet sowohl die Lösung von Expertenaufgaben, als auch die ‘intelligente’ Unterstützung des Experten bei seiner Arbeit. Ein Programm ist dann *wissensbasiert*, wenn das Wissen des Anwendungsbereiches deklarativ und explizit repräsentiert wird.“

Expertensysteme können auf unterschiedliche Weise klassifiziert werden. Eine gebräuchliche Klassifikation besteht darin, Expertensysteme domänenunabhängig und abstrahiert von dem jeweiligen Anwendungsgebiet anhand ihrer Problemklasse in unterschiedliche Kategorien einzuteilen. Bei der aufgabenorientierten Klassifikation werden

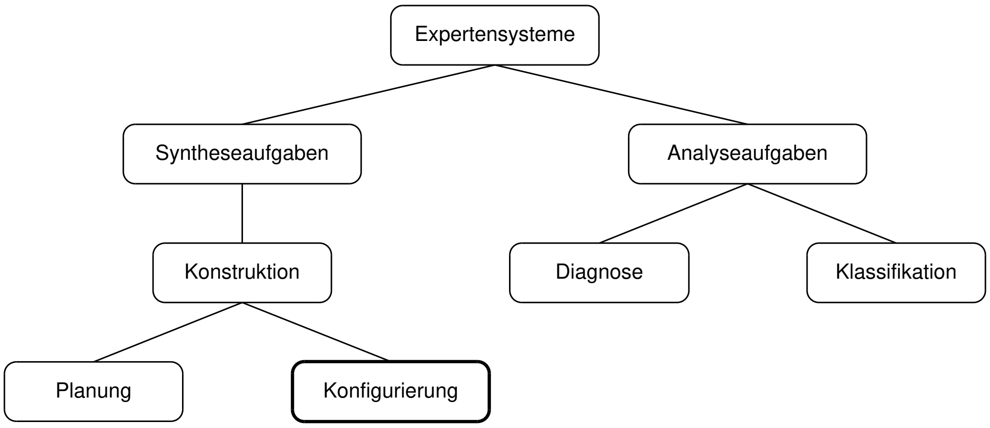


Abbildung 2.1: Einordnung der Konfigurierung

hauptsächlich zwei Gruppen von Expertensystemen unterschieden: Zum einen werden Systeme berücksichtigt, die das *Analyseproblem* behandeln, d. h. es werden existierende Objekte untersucht. Zum anderen gibt es dem *Syntheseproblem* gewidmete Expertensysteme, die sich mit der Erstellung neuer Objekte befassen. Überschneidungen wie auch die Reinformen sind dabei möglich. Konfigurierung ist den *Syntheseaufgaben* zuzuordnen. Sie wurden erst relativ spät von der Forschung angegangen und so mündete die Erforschung des Analyseproblems bereits in der erfolgreichen Entwicklung von Anwendungen wie *Diagnose*- und *Klassifikationssysteme*, als bei der Synthese noch Grundlagenforschung betrieben wurde (vgl. Weule 1993, S. 8).

Den Syntheseaufgaben lassen sich, wie in Abbildung 2.1 dargestellt, *Konstruktionsaufgaben* zuordnen, die wiederum in *Konfigurierungsaufgaben* und *Planungsaufgaben* unterteilt werden (vgl. Günter 1991b, S. 1). Konstruktionsaufgaben sind dadurch gekennzeichnet, dass im Verlauf der Konstruktion aus einer Reihe von Bauteilen bzw. Planschritten ein Gesamtkonzept, eine *Konfiguration* bzw. ein *Plan*, erstellt wird (vgl. Neumann 1991, S. 12):

Definition 2.1.1 (Konstruktionssystem)
Ein Konstruktionssystem ist ein Expertensystem, das beim Entwurf von Aggregaten aus Komponenten hilft und dabei Zielvorgaben sowie Expertenwissen verwendet.

Konfigurierung und Planung sind sich sehr ähnlich. Planung unterscheidet sich von Konfigurierung dadurch, dass für die Planung spezielle Formalismen und Verfahren für die Repräsentation von *Zeit* und deren Behandlung benötigt werden. Beispiele für Konstruktionsaufgaben sind (vgl. Neumann 1991, S. 13):

- Komponenten für ein Stromversorgungsaggregat mit vorgegebener Leistung auswählen,
- den Grundriss eines Gebäudes entwerfen,
- eine Abfolge von Schritten für ein Laborexperiment planen,

- ein Computersystem nach Kundenspezifikation konfigurieren,
- Möbel für ein Büro aussuchen und platzieren,
- Arbeitspläne für einen Produktionsprozess erstellen,
- Methoden für ein Bildverarbeitungssystem zur Qualitätsüberprüfung auswählen und konfigurieren.

Diese Arbeit beschränkt sich auf Konfigurierungsaufgaben in technischen Domänen. Eine Konfigurierungsaufgabe ist dadurch gekennzeichnet, dass eine Gesamtlösung, eine Konfiguration in einer Sequenz aus einzelnen Konfigurierungsschritten, aus separaten Einzelkomponenten (Konfigurierungsobjekte) zusammengefügt wird (vgl. Cunis und Günter 1991, S. 37). Mit einer Konfigurierungsaufgabe ist in diesem Kontext also *nicht* die Konfigurierung von bestehenden Anwendungen, Komponenten o. ä. gemeint, sondern vielmehr die Erstellung von neuen Systemen anhand bestimmter Randbedingungen. Konfigurierungsaufgaben sind überwiegend durch die folgenden Merkmale gekennzeichnet (vgl. Günter 1991b, S. 2; Neumann et al. 1987, S. 349 f.):

- *Großer Lösungsraum*
  Die Anzahl der prinzipiell möglichen Konfigurationen kann sehr groß sein.
- *Rücknahme von Entscheidungen*
  Während der Konfigurierung müssen z. T. Entscheidungen getroffen werden, die später nicht mehr haltbar sind und zurückgenommen werden müssen, um zu einer annehmbaren Lösung zu gelangen.
- *Hierarchisches Vorgehen*
  Eine Konfiguration ist in vielen Anwendungsbereichen eine Hierarchie von Komponenten, daher ist für den Konfigurierungsvorgang häufig ebenfalls eine hierarchische Gliederung zu empfehlen (z. B. *Top-Down*-Vorgehen).
- *Behandlung von Abhängigkeiten*
  Abhängigkeiten zwischen den Konfigurierungsobjekten sind von zentraler Bedeutung, denn darauf aufbauend wird die Konfigurierung durchgeführt. Eine effiziente und adäquate Repräsentation und Verarbeitung ist daher unerlässlich.

Konfigurierungssysteme in technische Domänen sind dadurch gekennzeichnet, dass sie zumeist gut strukturiertes Domänenwissen mit präzisen technischen Informationen aufweisen.[1] Der hohe Grad an Detailwissen führt dazu, dass das Ergebnis aus einer Vielzahl möglicher Lösungen bestimmt werden muss, was wiederum bedeutet, dass u. U. eine große Zahl möglicher (Lösungs-)Varianten berücksichtigt werden muss.

[1]Präzise formuliert wird von Rechenanlagen ausschließlich die Repräsentation der in dem Wissen enthaltenen Informationen in einer entsprechenden Repräsentationssprache verarbeitet (vgl. Broy 1992, S. 3 ff.).

2.2 Methoden zur Konfigurierung

In diesem Abschnitt wird nachfolgend eine Auswahl von Methoden zur wissensbasierten Konfigurierung vorgestellt. Sie unterscheiden sich z. T. sowohl in der Art der Wissensmodellierung als auch in der Weise, wie Konfigurierungsentscheidungen getroffen werden. Die vorgestellten Verfahren sind innerhalb von Konfigurierungsanwendungen eher selten in Reinform anzutreffen, häufig kommt eine Kombination der Methoden zum Einsatz. Übersichten hierzu sind in den Arbeiten von Günter und Kühn (1999); Kühn (2001); Sabin und Weigel (1998) und Stumptner (1997) zu finden.

2.2.1 Regelbasiertes Konfigurieren

Die Entstehung der ersten Expertensysteme war in den 80er Jahren stark von dem regelbasierten Ansatz dominiert. Expertensysteme wurden daher in den Anfängen teilweise als regelbasierte Systeme charakterisiert (vgl. Günter 1992, S. 44; Simon 1993, S. 265). Regelbasierte Expertensysteme nutzen zur Formulierung von Abhängigkeiten *assoziative* Regeln mit einem Bedingungs- und einem Aktionsteil in Form von

```
IF <Bedingungen> THEN DO <Aktionen>
```

Neben der Beschreibungsmöglichkeit von kausalen Abhängigkeiten zwischen Objekten, liegen die Stärken des regelbasierten Ansatzes in der Repräsentation und Evaluierung von heuristischen Abhängigkeiten.[2] Jede Regel ist eine unabhängige Wissenseinheit, die von einem domänenunabhängigen Regelinterpreter ausgeführt wird. Es wird zwischen *datenorientiertem* und *zielorientiertem* Inferenzmechanismus unterschieden (vgl. Günter 1992, S. 45 f.): Datenorientierte Regelsysteme prüfen anhand der vorliegenden Daten und des Bedingungsteils der Regeln, welche Regeln ausführbar sind (*Vorwärtsverkettung*). Zielorientierte Systeme erfassen den Aktionsteil der Regeln und vergleichen diese mit dem zu erreichenden Ziel. Alle Regeln, deren Aktionsteil das Ziel enthält, werden in einer Konfliktmenge zusammengefasst, aus der letztendlich eine Regel ausgewählt wird. Die Bedingungen der jeweiligen Regel sind wiederum die neuen Ziele (*Rückwärtsverkettung*). In einer Konfliktsituation, d. h. wenn mehrere Regeln die Anforderungen erfüllen, kann die Reihenfolge der Ausführung domänenunabhängig oder in Abhängigkeit vom Domänenwissen erfolgen. Eine domänenunabhängige Regelauswahl würde z. B. die aktuellste oder speziellste Regel verwenden. Für eine domänenabhängige Auswahl können z. B. Prioritäten, Bewertungsfunktionen oder Metaregeln die Auswahl unterstützen (vgl. Kühn 2001, S. 46).

In der Vergangenheit führte allerdings die intensive Nutzung von Regeln zur Wissensrepräsentation dazu, dass eine Reihe von Unzulänglichkeiten zu Tage traten. So findet in fast allen regelbasierten Systemen eine Vermischung von Domänenwissen und Kontrollwissen innerhalb der Regeln statt. Bei wachsendem Umfang der Systeme ergaben sich dadurch enorme Probleme bei der Wissensakquisition, der Sicherstellung der Konsistenz der Wissensbasis, der Modularität, der Adaptierbarkeit der Systeme, der Rücknahme von

[2] Den sog. „Daumenregeln“ der Experten (vgl. Günter 1992, S. 44).

```
ASSIGN-POWER-SUPPLY-1

IF:
THE MOST CURRENT ACTIVE CONTEXT IS ASSIGNING A POWER SUPPLY
AND AN SBI MODULE OF ANY TYPE HAS BEEN PUT IN A CABINET
AND THE POSITION IT OCCUPIES IN THE CABINET IS KNOWN
AND THERE IS SPACE IN THE CABINET FOR A POWER SUPPLY
AND THERE IS NO AVAILABLE POWER SUPPLY
AND THE VOLTAGE AND FREQUENCY OF THE COMPONENTS IS KNOWN

THEN:
FIND A POWER SUPPLY OF THAT VOLTAGE AND FREQUENCY
AND ADD IT TO THE ORDER
```

Abbildung 2.2: Regel aus dem XCon-System (vgl. Neumann 1988, S. 34)

Entscheidungen bei Konflikten und der Integration von Benutzer Anweisungen (vgl. Cunis und Günter 1991, S. 48; Günter 1991a, S. 108 f.; Günter 1992, S. 3, 54 ff. u. 209 f.; Günter und Kühn 1999, S. 49 f.; Neumann 1991, S. 15 ff.; Sabin und Freuder 1996a, S. 31; Sabin und Freuder 1996b, S. 156; Sabin und Weigel 1998, S. 44).

Der erste und wohl bekannteste Vertreter der Gattung der regelbasierten Expertensysteme ist das System R1/XCon. XCon war lange Zeit ein sehr erfolgreiches System zur Konfigurierung von VAX-Rechnern der Firma Digital Equipment Corporation (DEC) (vgl. Stumptner 1997, S. 111). Ein Beispiel für die Beschreibung einer Regel aus dem XCon-System ist in Abbildung 2.2 zu sehen. Die anfallenden Pflege- und Wartungsarbeiten der umfangreichen Regelbasis von XCon bereitete jedoch mit der Zeit zunehmend große Probleme (vgl. Abschnitt A.1, S. 260). Die Wissensbasis von XCon bestand 1989 aus ca. 11.500 Regeln (bezogen auf ca. 30.000 Objektbeschreibungen), von denen pro Jahr ca. 40–50% modifiziert werden mussten (vgl. Neumann 1991, S. 15; Günter 1992, S. 57). Verschiedene Ansätze zur Verfeinerung des Regelmechanismus, z. B. die Modularisierung der Wissensbasis durch das Zusammenfassen von Regeln zu sog. *Regelkontexten*,[3] brachte keine wirksame Abhilfe, da nun eine Vielzahl zusätzlicher Abhängigkeiten der Regeln untereinander verwaltet werden mussten, und Regeln zudem nicht mehr als unabhängige Wissenseinheit betrachtet werden konnten (vgl. Günter 1992, S. 48 ff.; Sabin und Weigel 1998, S. 44).

Aufgrund der Schwächen von regelbasierten Systemen sollte dieser Mechanismus in Expertensystemen nur lokale Anwendung finden und in Kombination mit anderen Methoden eingesetzt werden. Zur Steuerung von globalen Prozessen des Systems sind Regeln allenfalls bedingt geeignet (vgl. Günter 1991a, S. 92; Günter 1992, S. 57; Neumann 1991, S. 18).

[3] Regelmengen, die für sich genommen keine Beziehungen zu Regeln innerhalb anderer Regelkontexte besitzen.

2.2.2 Strukturbasiertes Konfigurieren

Moderne Konfigurierungswerkzeuge verwenden zur Repräsentation des Objektwissens der Domäne üblicherweise eine objektorientierte bzw. framebasierte Repräsentation. Sie ermöglicht die zusammengefasste Spezifikation der Objektstruktur, den Eigenschaften und möglichen Belegungen in einer sog. *Begriffshierarchie* innerhalb der Wissensbasis. Der von Günter (1992) vorgestellte strukturbasierte Ansatz definiert die Art des Vorgehens bei der Lösung des Konfigurierungsproblems anhand der Struktur der Konfigurierungsobjekte: „Beim strukturbasierten Konfigurieren orientiert sich der Problemlösungsvorgang an der Struktur des Domänenmodells“ (Kühn 2001, S. 47).

Die Konzepte stehen dazu innerhalb einer *Ontologie* über verschiedene Relationen zueinander in hierarchischer Beziehung (vgl. Noy und McGuinness 2001). Objektbeschreibungen werden in sog. *taxonomischen* und *partonomischen Hierarchien* klassifiziert, d. h. bestimmte Komponenten stehen vertikal über eine *is-a*-Relation und horizontal über eine *has-parts*-Relation zueinander in Beziehung. Diese Strukturierung des Wissens ermöglicht es, auf hohem Abstraktionsniveau allgemeine Beschreibungen der Objekte und deren speziellen Ausprägungen zu spezifizieren. Für den Konfigurierungsvorgang ermöglicht dies sowohl ein *Top-Down-* („Verfeinerung von Aggregaten, Dekomposition“) als auch ein *Bottom-Up*-Vorgehen („Aggregation von Komponenten“), orientiert an dem Aufbau der Komponentenstruktur (vgl. Cunis und Günter 1991, S. 45).

Für gewöhnlich werden während des Konfigurierungsverlaufs die Komponenten innerhalb der *Spezialisierungshierarchie* der Taxonomie über *is-a*-Relationen weiter spezialisiert, bis in der Hierarchie ein Blattkonzept erreicht ist, von dem aus keine weitere Spezialisierung möglich ist. Die Vermeidung von redundanten Beschreibungen durch Vererbungsmechanismen ermöglicht eine effiziente Verarbeitung und Wartung der Wissensbasis. Ebenso wichtig für die strukturbasierte Konfigurierung ist die kompositionelle Struktur der Partonomie. Über *has-parts*-Relationen stehen die Aggregate mit ihren Komponenten innerhalb einer *Zerlegungshierarchie* zueinander in Beziehung. Die Struktur der Lösung des Konfigurierungsproblems findet sich in ihrer abstraktesten Ausprägung bereits in der Struktur der Partonomie wieder. Auf Basis der hier beschriebenen Struktur werden die Aggregate und ihre Komponenten zusammengefügt (vgl. Günter 1992, S. 96 ff.).

Die strukturbasierte Konfigurierung ist ein leistungsfähiger Ansatz, mit dem sich aufgrund der vorhandenen Relationen innerhalb einer Begriffshierarchie während der Konfigurierung aus der vorhandenen Struktur des Domänenwissens Inferenzen hinsichtlich einer konsistenten Konfiguration bilden lassen. Eine wesentliche Annahme dieses Ansatzes ist allerdings die oben beschriebene Strukturiertheit der Domäne, wie sie z. B. im technischen Bereich häufig anzutreffen ist. Dies bedeutet, dass sich Domänen mit wenigen oder keinen Strukturinformationen entsprechend ungünstig oder gar nicht mit einem strukturbasierten Konfigurierungswerkzeug verarbeiten lassen (vgl. Günter 1991a, S. 110).

Der strukturbasierte Ansatz findet in Konfigurierungswerkzeugen häufig Anwendung, wenn auch oft implizit in Kombination mit anderen Verfahren. Das Konfigurierungswerkzeug EngCon (vgl. Hollmann et al. 2000) und seine Vorgänger PlaKon (vgl. Cunis et al. 1991) und KonWerk (vgl. Günter 1995a) sind klassische Beispiele für strukturbasiertes Konfigurieren.

2.2.3 Constraint-basiertes Konfigurieren

Constraints definieren Beschränkungen und Beziehungen zwischen Konfigurierungsobjekten bzw. zwischen den Eigenschaften (den Attributen) der Objekte. Mit Hilfe von Constraints können ungerichtete Abhängigkeiten repräsentiert und evaluiert werden. Constraints sind Teil des Domänenwissens und werden ebenfalls in der Wissensbasis spezifiziert. Sie dienen zum einen dazu, aus vorhandenen Eigenschaften spezifische Eigenschaften abzuleiten (*propagieren* von Constraints), und zum anderen zur Überprüfung der Konsistenz der aktuellen Konfiguration. Dies wird erreicht, indem Constraints in sog. *Constraint-Netzen* zusammengefasst werden, in denen sie über ihre Variablen, den *Constraint-Pins*, zueinander in komplexen Beziehungen stehen (vgl. Abschnitt 3.6.3, S. 39).

Zur Auswertung von Constraints gibt es eine Reihe von unterschiedlichen Verfahren, auf die ausführlich in Kapitel 4 und Kapitel 5 eingegangen wird. Ebenso gibt es mittlerweile eine große Anzahl von kommerziellen als auch frei verfügbaren Systemen und Werkzeugen zur Verarbeitung von Constraints, die diese Verfahren anwenden (vgl. Abschnitt 4.5, S. 65). Konfigurierungsaufgaben stellen allerdings generell einige besondere Anforderungen an ein Constraint-System. So wird das Constraint-Netz erst während des Konfigurierungsvorgangs inkrementell aufgebaut, d. h. die vollständige Anzahl der Constraints ist erst dann bekannt, wenn die Lösung vorliegt. Das Constraint-System sollte dementsprechend den inkrementellen Aufbau des Constraint-Netzes erlauben. Außerdem ist es je nach Anwendungsdomäne erforderlich, dass das Constraint-System in der Lage ist, unterschiedliche Arten von Constraints (z. B. symbolische, numerische, extensionale, funktionale, usw.) zu verarbeiten (vgl. Günter und Kühn 1999, S. 52).[4]

Ein weiteres Problem ist, dass die Komplexität von Constraint-Problemen *NP-vollständig* ist, wenn die Propagation der Constraints globale Konsistenz innerhalb des Constraint-Netzes sicherstellen soll. Häufig kann daher z. B. aufgrund der Größe einer Domäne nicht jeder einzelne Wert propagiert werden, sondern nur jeweils die Ober- und Untergrenzen eines Attributs bzw. Parameters. Das Constraint-System sollte demnach einen Kompromiss zwischen Effizienz und Qualität erlauben, indem entsprechend dem Anwendungsfall, das Verfahren mit der benötigten Korrektheit bei der Berechnung der Constraints Anwendung findet.

Weil allgemein anerkannt ist, dass Constraints und constraint-basierte Lösungsverfahren zur flexiblen Beschreibung von Abhängigkeiten und Auflösung selbiger besonders geeignet sind, werden Konfigurierungswerkzeuge vielfach als „constraint-basiert“ vermarktet, obwohl dies streng genommen nicht zutrifft. Zum Teil besitzen derartige Werkzeuge keinen Propagationsmechanismus sondern lediglich eine constraint-basierte Problemspezifikation zur Konsistenzüberprüfung, andere Werkzeuge ermöglichen lediglich eingeschränkte Möglichkeiten zur Nutzung von Constraints während des Konfigurierungsvorgangs (vgl. John und Geske 1999a, S. 2).

Ein verfügbares Constraint-System, welches die Anforderungen für ein Konfigurierungswerkzeug in allen Belangen erfüllt, ist derzeit nicht bekannt. Aufgrund dessen sind Konfigurierungswerkzeuge wie EngCon auf die Implementierung von Eigenlösun-

[4] Für weitere Anforderungen, speziell für das Konfigurierungswerkzeug EngCon, vgl. Abschnitt 3.8 auf Seite 44 ff.

gen angewiesen, die z. T. auf die Anbindung externer Komponenten zurückgreifen (vgl. Abschnitt 3.6, S. 33). Interessante Ansätze bezogen auf Konfigurierungsaufgaben sind *Dynamic Constraint Satisfaction* (vgl. Mittal und Falkenhainer 1990) und *Generative Constraint Satisfaction* (vgl. Stumptner et al. 1998) sowie das constraint-basierte System ConBaCon (vgl. John und Geske 2002), in dem eine Konfigurierung mit Hilfe von bedingter Constraint-Propagation durchgeführt wird (vgl. Sabin und Weigel 1998, S. 46 f.; Günter und Kühn 1999, S. 52 u. 59).

2.2.4 Ressourcenbasiertes Konfigurieren

Ressourcen sind abstrakte Leistungen, die Beziehungen zwischen Konfigurierungsobjekten beschreiben (vgl. Kühn 2001, S. 47). Der ressourcenbasierte Ansatz beruht auf der Annahme, dass Komponenten in einem technischem System eingesetzt werden, weil sie etwas anbieten, was von dem System als Ganzes oder von einzelnen Teilen des Systems benötigt wird (vgl. Günter und Kühn 1999, S. 53). Die Schnittstellen zwischen den Komponenten werden über die ausgetauschten *Ressourcen* spezifiziert. Selbiges betrifft die Beziehung zwischen dem System und das es umgebende Umfeld.

Komponenten machen bestimmte Ressourcen verfügbar und „konsumieren“ wiederum andere Ressourcen. Die Aufgabenstellung bei Beginn der Konfigurierung ergibt sich aus initialen Ressourcenforderungen. Diese werden in einem iterativen Konfigurierungsprozess durch Komponenten, die entsprechende Ressourcen anbieten, ausgeglichen. Der Konfigurierungsverlauf ist somit ein Prozess des Bilanzierens und Ausgleichens von Ressourcenforderungen (vgl. Heinrich und Jüngst 1993; Jüngst und Heinrich 1998).

Technische Ressourcen sind z. B. der Stromverbrauch und die Speicherkapazität von Komponenten. Beispiele für wirtschaftliche Ressourcen sind Preis und Wartungsaufwand. Die Systeme Cosmos (vgl. Günter et al. 1999) und KIKon (vgl. Emde et al. 1996) sind Beispiele für den Einsatz ressourcenbasierter Konfigurierungswerkzeuge. Der Vorteil der ressourcenbasierten Modellierung von Konfigurierungsproblemen wird in dem geringeren Wartungsaufwand gesehen, der sich ergibt, weil abstrakte Ressourcen eine längere „Lebensdauer“ aufweisen, als konkrete Komponenten (vgl. Günter et al. 1999, S. 62).

2.2.5 Fallbasiertes Konfigurieren

Die fallbasierte Konfigurierung ist dadurch gekennzeichnet, dass dem System bereits vor der eigentlichen Konfigurierung Wissen über bereits gelöste Konfigurierungsprobleme innerhalb einer „Fallbibliothek“ bekannt ist. Die Annahme dabei ist die, dass ähnliche Probleme zu ähnlichen Lösungen führen werden.

Dazu kann ein Preprozessing zur Strukturierung der *Fallbasis* durchgeführt werden. Andere Systeme arbeiten einfach auf der unstrukturierten Menge von fallbasiertem Wissen. Es ist nicht einmal notwendig, jeweils den gesamten Fall zu betrachten, da auch Teillösungen einbezogen werden können. Generell gibt es zwei Vorgehensweisen zur Auswahl von fallbasiertem Wissen (vgl. Günter und Kühn 1999, S. 53):

- Einen hinreichend ähnlichen Fall suchen und das Ergebnis adaptieren (*transformational analogy*).

- Auswahl des fallbasierten Wissens für jeden einzelnen Schritt. In diesem Fall kann die Fallbasis als Kontrollwissen (vgl. Abschnitt 3.5, S. 29) der Konfigurierung dienen (*derivational analogy*).

Unabhängig von der Auswahl muss anschließend eine Adaptierung des fallbasierten Wissens an das aktuelle Problem vorgenommen werden. Diese Anpassung ist für gewöhnlich relativ umfangreich. Zudem hat das fallbasierte Vorgehen einige gravierende Nachteile. So werden von fallbasierten Systemen üblicherweise relativ „konservative“ Lösungen angeboten, da die Lösungen aufgrund bekannter Fälle zustande gekommen sind. Aufgrund dessen kann das System i. A. auch keine kausale Erklärung über das Zustandekommen einer bestimmten Lösung liefern.

2.2.6 Verhaltensbasiertes Konfigurieren

Als *verhaltensbasierte Konfigurierung* wird die Erweiterung der regel-, struktur-, constraint-, ressourcen- oder fallbasierten Konfigurierung um Verhaltenswissen bzgl. des zu konfigurierenden Systems bzw. der Konfigurierungsobjekte bezeichnet. Das Wissen über das Verhalten von Objekten kann z. B. in Form von Zustandsdiagrammen repräsentiert werden (vgl. Kühn 2000, S. 92 ff.).

„Unter dem *verhaltensbasierten Konfigurieren* verstehen wir einen Spezialfall des Konfigurierens, wobei Wissen über das Verhalten der Konfigurierungsobjekte oder des Gesamtsystems zur Problemlösung beiträgt“ (Kühn 2001, S. 46). Als das Verhalten eines Objektes bzw. eines Systems werden die „Zustandsänderungen seiner veränderlichen Eigenschaften über die Zeit“ bezeichnet. Während der verhaltensbasierten Konfigurierung wird nicht nur strukturelles Wissen berücksichtigt, sondern auch das Systemverhalten der Komponenten. Mittel der verhaltensbasierten Konfigurierung sollten in Fällen eingesetzt werden, in denen das Verhalten des Systems wichtiger ist als der strukturelle Aufbau, oder selbiges nicht direkt aus letzterem abgeleitet werden kann.

Das Wissen über das Verhalten von Objekten lässt sich für die verschiedenen Verfahren auf unterschiedliche Weise adaptieren. Verhaltensinformationen können bei regelbasierten Systemen z. B. Wissenseinheiten sein, die innerhalb von Regeln sowohl im Bedingungs- als auch im Aktionsteil Anwendung finden können. Bei der strukturbasierten Konfigurierung wird das Verhaltenswissen als Eigenschaft von Objekten der Begriffshierarchie modelliert.[5] Die constraint-basierte Konfigurierung sieht ein Constraint-System vor, dass eine spezielle Klasse von „Verhaltens-Constraints“ zur Verfügung stellt. Ebenso lässt sich für das ressourcenbasierte Konfigurieren das Verhaltenswissen als Ressourcen-Typ repräsentieren, oder entsprechend für fallbasiertes Konfigurieren als Index zum Wiederauffinden von Fällen innerhalb der Fallbasis anhand des Verhaltens selbiger. Zusätzlich kann Verhaltenswissen bei der Übernahme und Anpassung von Fällen unterstützend eingesetzt werden.

[5] Für die Kombination von strukturbasierter Konfigurierung mit Mechanismen zur Auswertung von Verhaltenswissen wird von Kühn (2003) eine eigene Methodik namens ABACUS (*A Behavior-Based Configuration Approach Using State Charts*) eingeführt.

2.3 Konfigurierungswerkzeuge

Im Folgenden werden eine Reihe von bestehenden Konfigurierungswerkzeugen und die jeweils eingesetzten Methoden benannt. Die vorgestellten Systeme stellen lediglich eine exemplarische Auswahl dar. Eine ausführliche Beschreibung der einzelnen hier genannten Systeme ist dem Anhang A auf Seite 260 ff. zu entnehmen.

Zunächst ist eine Reihe historischer Systeme zu erwähnen, deren Eigenschaften für die Entwicklung von EngCon und dessen Vorgänger z. T. eine besondere Bedeutung hatten. Die erste Konfigurierungsanwendung überhaupt war Anfang der 80er Jahre das bereits erwähnte System XCon, der klassische Vertreter des regelbasierten Ansatzes. Die große Anzahl Regeln zur Konfigurierung einer Vielzahl komplexer Objekte führte schließlich zu einem erheblichen Software-Wartungsproblem (vgl. Neumann 1988, S. 34). Das System Siconfex, ebenfalls in den 80er Jahren entwickelt, war bereits mit einer hochstrukturierten Wissensbasis ausgestattet, die Schemata und Vererbungsmechanismen unterstützte. Inferenzmechanismus war weiterhin ein, wenn auch erweiterter, Regelmechanismus. Eine weitere Strukturierung des Domänenwissens wurde in den Systemen MMC-Kon und All-Rise vorgenommen: Neben *is-a* gibt es hier auch bereits *has-parts* Relationen innerhalb von Spezialisierungs- und Zerlegungshierarchien. In All-Rise werden Abhängigkeiten zudem bereits durch Constraints repräsentiert und propagiert. Alle genannten Systeme wurden zum Lösen jeweils spezifischer Konfigurierungsaufgaben entworfen und eingesetzt (vgl. Neumann 1991).

Moderne Konfiguratoren besitzen eine domänenunabhängige Wissensrepräsentation. Sofern sie in überschaubaren Domänen eingesetzt werden, verwenden sie aus Gründen der Akzeptanz zumeist relativ einfache Inferenzmechanismen. Der CAS-Konfigurator und Cameleon EPOS setzen sowohl Regeln als auch Entscheidungsbäume zur Steuerung des Konfigurierungsprozesses ein. Im camos.Configurator werden sog. „passive“ Constraints verwendet. Passive Constraints sind Regeln, die bidirektional ausgewertet werden, jedoch keinen Einfluss auf den Kontrollfluss haben (vgl. Günter et al. 1999). Ebenfalls eine eingeschränkte Art Constraints, sog. „gerichtete“ Constraints, nutzt das Konfigurierungswerkzeug Comix (vgl. Sutschet 2001). Der Konfigurator salesPLUS (vgl. Yu und Skovgaard 1998) von Baan verfügt über ein eigens implementiertes Constraint-System, welches die Verarbeitung spezieller Constraint-Typen zur Unterstützung des Konfigurierungsvorgangs ermöglicht. Die Konfigurierung selbst zeichnet sich allerdings lediglich durch die konsistente Anpassung bestehender Standard-Produktmodelle aus. Bei der Sales Configuration Engine (SCE) von SAP kommt ein *Truth Maintenance System* (TMS) zum Einsatz. Ziel ist es, dadurch ein wissensbasiertes Backtracking zum Auffinden von Fehlerursachen bei Konfigurierungskonflikten und die Interpretation von Design-Entscheidungen zu ermöglichen (vgl. Günter und Kühn 1999, S. 59).[6] Die Systeme Cosmos (vgl. Günter et al. 1999) und KIKon (vgl. Emde et al. 1996) sind Beispiele für ressourcenbasierte Konfigurierungswerkzeuge. Sie bieten für Domänen, deren Objekte sich als Ressourcen modellieren lassen, eine Abstraktion von einzelnen Komponenten auf deren Ressourcenforderung bzw. -angebot.

[6] Bei der Rücknahme von Entscheidungen lassen sich auf diese Weise Abhängigkeiten identifizieren und ggf. weitere Konfigurierungsschritte ebenfalls zurücknehmen.

Bei modernen Konfigurierungsanwendungen, die zur Bearbeitung umfangreicher Domänen mit komplexen Abhängigkeiten entworfen wurden, setzt man überwiegend auf erweiterte constraint-basierte Ansätze, welche die Dynamik des Konfigurierungsverlaufs angemessen unterstützen. So besitzt der TACTON CONFIGURATOR ein dynamisches Constraint-System, welches ein während der Konfigurierung anwachsendes Constraint-Netz unterstützt (vgl. Orsvärn und Axling 1999). Die Constraint-Lösungsmechanismen von LAVA (vgl. Fleischanderl et al. 1998) und ILOG CONFIGURATOR (vgl. ILOG 2001) bzw. JCONFIGURATOR (vgl. ILOG 2003) implementieren *Generative Constraint Satisfaction*, ein ebenfalls dynamischer Constraint-Formalismus, der den Anforderungen zur Behandlung von komplexen Konfigurierungsproblemen gerecht wird (vgl. Abschnitt 4.4, S. 57).

Gegenstand aktueller Forschung ist das Konfigurierungswerkzeug CONBACON (vgl. John und Geske 2002). Der Schwerpunkt von CONBACON liegt auf einem Constraint-System, welches die effiziente Behandlung einer Rekonfigurierung nach Erhalt von Änderungswünschen des Anwenders ermöglicht. Das Projekt CAWICOMS (vgl. Ardissono et al. 2002), ebenfalls aktueller Forschungsgegenstand, hat zum Ziel, ein B2B-Framework zur verteilten Produktkonfigurierung von Waren und Dienstleistungen anzubieten. Dafür kommen neben dem JCONFIGURATOR als Inferenz-Engine moderne Web-Technologien zum Einsatz.

Sowohl ENGCON (vgl. Hollmann et al. 2000) als auch dessen Vorläufer PLAKON (vgl. Cunis et al. 1991) und KONWERK (vgl. Günter 1995a) sind ebenfalls für variantenreiche Konfigurierungsszenarien geeignet. Sie sind in erster Linie strukturbasierte Konfigurierungswerkzeuge. Der Schwerpunkt dieser Systeme liegt daher auf einem begriffshierarchie-orientierten Kontrollmechanismus. Zusätzlich kommen weitere Inferenzmechanismen zum Einsatz, wie z. B. ein ausgereiftes Constraint-System. Auf das Konfigurierungssystem ENGCON wird in dem folgenden Kapitel ausführlich eingegangen.

Kapitel 3

Wissensbasierte Konfigurierung am Beispiel von EngCon

Any sufficiently advanced technology
is indistinguishable from magic.

Arthur C. Clarke
(Clarke's Third Law)

In diesem Kapitel wird ein Überblick über das strukturbasierte Konfigurierungswerkzeug EngCon gegeben. Es werden die konzeptionelle Architektur sowie die unterschiedlichen Wissensarten beschrieben. Außerdem folgt eine Übersicht und eine Diskussion zum Constraint-System sowie die Anforderungen von EngCon bzgl. der einzusetzenden Constraint-Lösungsverfahren.

3.1 Übersicht

Im Folgenden wird das am Technologie-Zentrum Informatik (TZI)[1] des Fachbereichs Mathematik und Informatik der Universität Bremen[2] entwickelte strukturbasierte Konfigurierungssystem EngCon beschrieben. EngCon ist ein Akronym für *Eng*ineering & *Con*figuration und ist, wie die meisten Konfigurierungssysteme, für den technischen Bereich zugeschnitten. Das System ermöglicht die Unterstützung von Ingenieuren bei Konstruktionsaufgaben z. B. zur Angebotserstellung und -validierung. EngCon kann überall dort eingesetzt werden, wo hochstrukturiertes Wissen über Aggregate und Komponenten vorliegt (vgl. Hollmann et al. 2000).

Der Schwerpunkt von EngCon liegt im Gegensatz zu z. B. XCon nicht auf Inferenzen, die aufgrund von Expertenregeln gebildet werden („regelhafte Expertise“), sondern

[1] `http://www.tzi.de`
[2] `htpp://www.informatik.uni-bremen.de`

auf Inferenzmechanismen, die aufgrund der wissensbasierten Architektur des Systems abgeleitet werden können. Als wissensbasierte bzw. hybride wissensbasierte Systeme werden nach Schlingheider (1994) Systeme bezeichnet, die eine deklarative Wissensrepräsentation erlauben, d. h. es kommen eine oder mehrere der unter Abschnitt 2.2 auf Seite 11 ff. vorgestellten Formen der Wissensrepräsentation zum Einsatz. Ein Vorteil von solchen Systemen ist, dass sich der Wissensingenieur beim Erstellen der Wissensbasis auf den Kern seiner Arbeit konzentrieren kann:

> „Der Arbeitsinhalt beim Kodieren der Wissensdomäne umfasst durch die von diesen Systemen angebotene Unterstützung weniger die Formulierung von Kontrollstrukturen zum Feuern von Regeln oder dergleichen. Es kann vielmehr verstärkt der eigentlichen Modellierung des Wissens mit Regeln, Funktionen oder Objektstrukturen Rechnung getragen werden“ (Schlingheider 1994, S. 55).

Eine Besonderheit von EngCon ist der inkrementell und interaktiv verlaufende Konfigurierungsprozess. Am Ende einer strukturbasierten Konfigurierung steht immer genau eine Lösung. Im Gegensatz zu anderen Konfigurierungswerkzeugen, die häufig eine Breitensuche vornehmen und abschließend alle gefundenen Lösungen präsentieren, findet während der Konfigurierung mit EngCon eine benutzergesteuerte Tiefensuche statt, mit dem Ziel, interaktiv eine für den Benutzer geeignete Lösung zu finden.

3.2 Historie

Das wissensbasierte Konfigurierungswerkzeug EngCon wurde im Rahmen des Projekts RAPA II der Firma Lenze AG[3] am TZI entwickelt. Das TZI ist ein Institut der Universität Bremen für den Technologieaustausch zwischen Wissenschaft und Industrie. EngCon ist eine Weiterentwicklung des im Rahmen des Verbundprojekts ProKon (*Pro*blemspezifische Werkzeuge für die wissensbasierte *Kon*figurierung) unter Koordinierung der Universität Hamburg entwickelten Vorgängers KonWerk (*Kon*figurierungs *Werk*zeug). In diesem Vorgängerprojekt wurden die Methoden der strukturbasierten Konfigurierung entwickelt und ein Prototyp in Common Lisp bzw. CLOS (*Common Lisp Object System*) implementiert. Dieser Prototyp wurde bei zahlreichen Anwendern praktisch erprobt und beständig weiterentwickelt (vgl. Günter 1995a, b). KonWerk beruht wiederum auf Ergebnissen aus dem Verbundvorhaben TEX-K, in dessen Rahmen ein „Expertensystemkern für *Pla*nungs- und *Kon*figurierungsaufgaben in technischen Domänen“ namens PlaKon ebenfalls hauptsächlich an der Universität Hamburg entwickelt und gleichfalls eine prototypische Implementierung in Lisp umgesetzt wurde (vgl. Cunis und Günter 1991).

Die Notwendigkeit für die Entwicklung eines eigenen Expertensystemkerns ergab sich daher, dass die Anforderungen für ein Expertensystem für Konstruktionsaufgaben von kommerziellen Systemen nur teilweise oder gar nicht erfüllt wurden. Um Expertensysteme in Konstruktionsdomänen entwickeln zu können, war es erforderlich, einen spezialisierten Expertensystemkern zu entwickeln. Der Schwerpunkt der Entwicklung lag verhältnismäßig stark auf Konfigurierung und betont weniger die Aspekte der Planung.

[3] `http://www.lenze.de`

Um eine möglichst unkomplizierte Einbettung in moderne Umgebungen zu gewährleisten, wurde der am TZI entwickelte Forschungsprototyp EngCon V0, auf den sich die weiteren Ausführungen dieser Arbeit beziehen, in Java 2 implementiert (vgl. Hollmann et al. 2000). Die Entwicklung erfolgte mit Unterstützung der Lenze AG und einer Ausgründung aus dem TZI, der encoway GmbH & Co. KG[4], einem Unternehmen der Lenze-Gruppe, sowie dem Hamburger Informatik Technologie-Center e. V.[5] (HITeC). Durch encoway wurde der Protoyp in ein kommerzielles Produkt überführt, dem Drive Solution Designer (DSD). Im DSD findet der Expertensystemkern EngCon V1 in Form der EngCon Configuration Suite (ECS) zur Konfigurierung von Motoren und Antriebstechnik (für Druck- und Sortiermaschinen, etc.) Verwendung (vgl. Ranze et al. 2002). Die zu konfigurierenden Antriebslösungen weisen mit ca. 10^{23} unterschiedlichen Produkt-Kombinationen eine sehr hohe Variantenvielfalt auf. Zurzeit wird der DSD von ca. 300 Vertriebsingenieuren der Lenze AG zur Planung und Auslegung von Antriebssystemen genutzt. Durch den Einsatz der Software wird die Zeit für eine Angebotserstellung stark reduziert, zudem vermindert sie deutlich die Fehlerquote bei den erzeugten Angeboten.

Für diesen äußerst erfolgreichen und innovativen Einsatz von Methoden der Künstlichen Intelligenz innerhalb eines marktreifen Produkts wurde der Drive Solution Designer von der *American Association for Artificial Intelligence*[6] mit dem Preis „Innovative Applications of Artificial Intelligence Award 2002“ ausgezeichnet (vgl. Campus Press 2002). Sowohl der Forschungsprototyp EngCon V0, als auch der DSD mit EngCon V1, sowie die Erweiterung eben dieser um neue Funktionalitäten ist Thema zahlreicher Veröffentlichungen (vgl. Arlt et al. 1999; Günter et al. 2001; Hollmann et al. 2000; Krebs et al. 2003; Ranze et al. 2002) und Diplomarbeiten (vgl. Bonar 2003; Krebs 2002; Werres 2002).

3.3 Architektur

Die Architektur von EngCon setzt sich, stark vereinfacht, aus drei wesentlichen Komponenten zusammen (siehe Abbildung 3.1 auf der nächsten Seite): der **Wissensbasis**, dem eigentlichen **Konfigurator** und einer **GUI** (*Graphical User Interface*). Die Wissensbasis ist eine deklarative Beschreibung. Sie ist nicht Teil des Programm-Codes und damit flexibel austauschbar, d. h. für jede Anwendung kann spezielles Wissen definiert und eingebunden werden. EngCon ist dadurch domänenunabhängig und kann durch einen Austausch der Wissensbasis beliebige variantenreiche Komponenten konfigurieren. Das Wissens wird in drei Klassen unterteilt:

- Das **konzeptuelle Wissen** beinhaltet das Domänenwissen in Form von Konzepten und deren Attribute, sowie Relationen zu anderen Konzepten.
- Das **Kontrollwissen** enthält das Wissen von Experten, wie und in welcher Reihenfolge eine Komponente zusammenzusetzen ist. Durch das hier abgelegte Wissen werden Benutzer in die Lage versetzt, qualitativ hochwertige Konfigurierungen

[4] `http://www.encoway.de`
[5] `http://www.hitec-hh.de`
[6] `http://www.aaai.org`

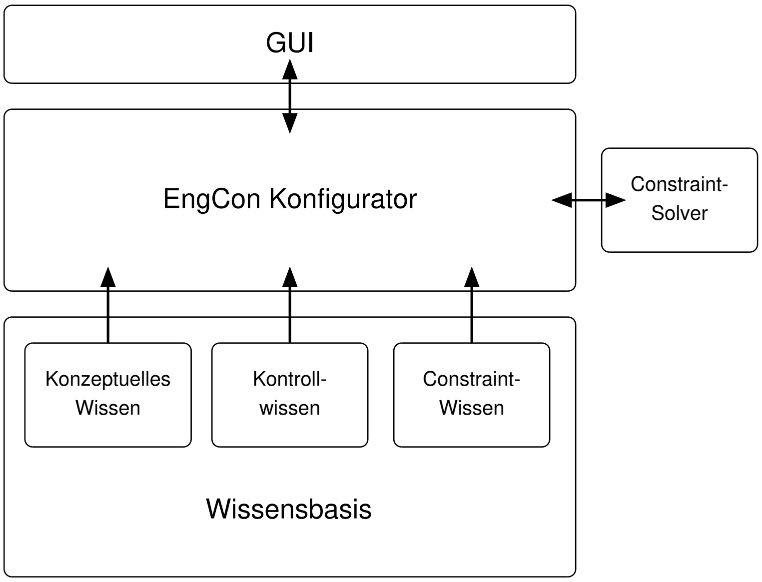


Abbildung 3.1: Architektur von EngCon (vgl. Ranze et al. 2002, S. 847)

durchzuführen, ohne über eigenes, entsprechend tiefgreifendes, Wissen zu verfügen. Der Benutzer wird vom System je nach eigenem Kenntnisstand geführt. Dazu lassen sich verschiedene Benutzerklassen und entsprechende Bearbeitungsverfahren, wie z. B. Default-Werte, definieren.

- Das **Constraint-Wissen** ermöglicht es durch Constraints komplexe Abhängigkeiten zwischen Konzepten zu modellieren, die sich mittels taxonomischer Relationen im Konzeptwissen nicht ohne weiteres oder gar nicht repräsentieren ließen.

Bei der Wissensmodellierung ist zu beachten, dass für das vorhandene Wissen die jeweils adäquate Modellierungsform Verwendung findet, um eine effiziente Verarbeitung zu gewährleisten. So ist es z. B. zwar möglich, eine Unterscheidung von schnellen und langsamen CD-ROM-Laufwerken anhand von Constraints vorzunehmen. Nahe liegender und weniger komplex ist jedoch, die entsprechende Struktur mit einer taxonomischen Relation im Konzeptwissen zu modellieren.

Die Komponente zum Auflösen von Constraints, der derzeit verwendete **Constraint-Solver**, ist als unabhängiges Modul an den EngCon Konfigurator angebunden. Das Ziel dieser Arbeit ist es, die derzeitige Anbindung durch eine flexible Lösung für unterschiedliche Constraint-Solver zu ersetzen, die am Beispiel von EngCon den inkrementellen Konfigurierungsprozess der strukturbasierten Konfigurierung unterstützt.

3.4 Konzeptuelles Wissen

Um die Objekte unterschiedlicher Domänen konfigurieren zu können, ist die Verwendung eines umfassenden, problemunabhängigen Modellierungsmechanismus notwendig. Eng-

Con verwendet eine eigene Begriffshierarchie zur deklarativen Definition des Domänenwissens. In der Begriffshierarchie werden mögliche Konfigurierungsentscheidungen durch *is-a-* und *has-parts*-Beziehungen definiert, allerdings ohne eine Reihenfolge vorzugeben. Dies erlaubt die Trennung zwischen dem im Domänenmodell enthaltenen Domänenwissen und dem Ablaufwissen des Experten. Die dadurch gewonnene Flexibilität unterscheidet strukturbasierte Systeme (vgl. Abschnitt 2.2.2, S. 13) wie EngCon von klassischen, regelbasierten Systemen wie z. B. XCon, die beinahe zwangsläufig eine Vermischung der Wissensarten mit sich bringen (vgl. Abschnitt 2.2.1, S. 11). Der in EngCon eingesetzte Mechanismus zur Ablaufkontrolle ist „modellgetrieben“, d. h. er „interpretiert“ das Domänenmodell (vgl. Cunis 1991, S. 59).

Die praktischen Beispiele, die während der Ausführungen in dieser Arbeit angegeben werden, beziehen sich i. d. R. auf die in der Vergangenheit im Bereich der Konfigurierung für diese Zwecke recht häufig genutzte „PC-Domäne“ (vgl. z. B. Mittal und Frayman 1989). Sie wurde bereits während der Entwicklung des EngCon-Prototypen am TZI und dessen Vorgänger (vgl. Günter 1995b, S. 77 ff.), sowie bei der Erstellung mehrerer Diplomarbeiten (vgl. Bonar 2003; Krebs 2002) zu Testzwecken eingesetzt. In einer Beispiel-Wissensbasis wird definiert, wie aus zahlreichen Einzelkomponenten ein PC-System konfiguriert werden kann. Die in diesem Szenario modellierte Domäne ist allgemein bekannt und hinreichend komplex, um beispielhaft die Problematik und die Funktion der in dieser Arbeit vorgestellten Verfahren aufzeigen zu können.

3.4.1 Begriffshierarchie

Die Begriffshierarchie enthält die konzeptuelle Beschreibung aller Objekte und Abhängigkeiten der Anwendungsdomäne. Die Organisation des Wissens erfolgt in Hierarchien von Konzepten mit Hilfe von zwei sich ergänzenden Relationen. Intern werden die Konzepte als *Frame* repräsentiert, dessen *Slots* die Eigenschaften der Konzepte aufnehmen. Die Beschreibung der framebasierten Wissenshierarchie erfolgt mittels *is-a-* und *has-parts*-Beziehungen. Die Struktur der Begriffshierarchie wird auch als *objektorientiert* bezeichnet, weil sie wesentliche Merkmale (Vererbung und Aggregation) der Objektorientierung aufweist (vgl. Abschnitt 2.2.2, S. 13).[7]

Die Begriffshierarchie beschreibt durch Konzepte und Relationen abstrakt alle in der Domäne möglichen bzw. zulässigen Konfigurationen und definiert damit den Suchraum (vgl. Günter 1992, S. 117). Weil die Begriffshierarchie von EngCon die Metarepräsentation aller zulässigen Konfigurationen darstellt, wird sie auch als *Ontologie* bezeichnet (vgl. Noy und McGuinness 2001). Ein Ausschnitt aus einer Beispiel-Begriffshierarchie ist in Abbildung 3.2 auf der nächsten Seite zu sehen. Alle Teilkonfigurationen und Komponenten können vom Wurzelknoten über *is-a-* und *has-parts*-Beziehungen erreicht werden. In einem *Zielkonzept* wird der Wurzelknoten der Konfiguration definiert. Das jeweilige Zielkonzept, von denen es beliebig viele in der Wissensbasis geben kann, wird zu Beginn der Konfigurierung ausgewählt. Mit der Auswahl des Zielkonzepts wird der weitere Ablauf

[7] Einen Mechanismus ähnlich den „Methoden“ aus der objektorientierten Programmierung gibt es freilich nicht.

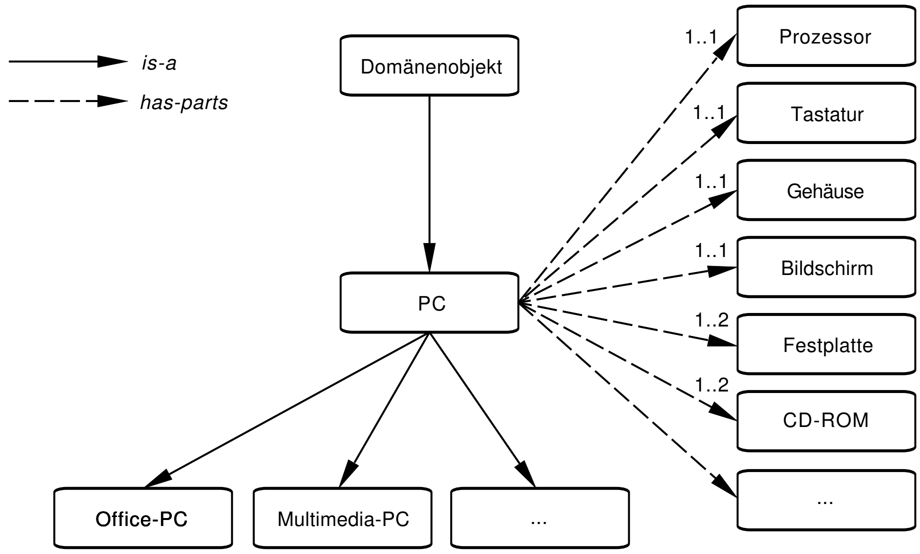


Abbildung 3.2: Ausschnitt aus der Beispiel-Begriffshierarchie (vgl. Günter 1995b, S. 78)

anhand der Struktur des definierten Wurzelknotens festgelegt (siehe Abbildung 3.3 auf der gegenüberliegenden Seite).

Für die Begriffshierarchie wird die *closed world assumption* (CWA) vorausgesetzt, d. h. es wird angenommen, dass mit der Begriffshierarchie ein vollständiges Abbild der Anwendungsdomäne vorliegt. Weitere Objekte z. B. sind nur durch eine Erweiterung des statischen Wissens zur Laufzeit möglich. Obwohl die CWA sehr einschränkend ist, so ist sie für technische Domänen angemessen, weil dort die Anzahl der Objekte begrenzt ist, und alle Objekte und Eigenschaften bekannt sein müssen, um eine Konfigurierung durchführen zu können (vgl. Cunis und Günter 1991, S. 44).

3.4.2 Konzeptuelle Hierarchie

Die Objekte, die in einer *is-a*-Beziehung (Generalisierung) zueinander stehen, beschreiben eine konzeptuelle oder auch *taxonomische* Hierarchie. Innerhalb einer Taxonomie werden Typen von Objekten definiert. Dazu erfolgt die Definition von Konzepten und Unterkonzepten (Spezialisierungen). Eigenschaften der Konzepte (Attribute und Relationen) werden innerhalb der Spezialisierungshierarchie an die Unterkonzepte vererbt. Diese Hierarchie ist eine Baumstruktur und schließt daher Mehrfachvererbung ausdrücklich aus.[8]

[8] In den Vorgängern von EngCon, PlaKon und KonWerk, ist mittels eines *Sichtenkonzepts* eine Erweiterung der Begriffshierarchie entwickelt und implementiert worden, die für die Konfigurierung dasselbe wie Mehrfachvererbung leistet. Mittels *Sichten* ist es möglich Objekte der Begriffshierarchie unter verschiedenen Gesichtspunkten zu betrachten, und sie z. B. in der jeweiligen Sicht unabhängig von anderen Sichten zu spezialisieren. Dabei ist im Gegensatz zur Mehrfachvererbung für jede Sicht eine eindeutige Vererbungslinie der Objekteigenschaften gewährleistet (vgl. Cunis 1991, S. 71 ff.; Günter 1995b, S. 117 ff.). Dieses Sichtenkonzept in der Begriffshierarchie wurde für EngCon bisher noch nicht implementiert.

```
(def-do
    :name PC
    :oberkonzept domaenenobjekt
    :parameter ((Preis [0.0 inf])
                (Icon "pc" (non-config true)))
    :relationen ((hat-komponente {[(ein PC_Komponente) 7 16] :=
                                  [(ein Gehäuse) 1 1]
                                  [(ein Mainboard) 1 1]
                                  [(ein Prozessor) 1 1]
                                  [(ein Speicher) 1 3]
                                  [(ein Netzwerkkarte) 0 2]
                                  [(ein VGA_Karte) 1 2]
                                  [(ein TV_Karte) 0 1]
                                  [(ein Soundkarte) 0 1]
                                  [(ein Festplatte) 1 2]
                                  [(ein CD_Rom) 1 2]})
                 (hat-peripherie {[(ein Peripherie_Komponente) 3 8] :=
                                  [(ein Monitor) 1 2]
                                  [(ein Maus) 1 1]
                                  [(ein Tastatur) 1 1]
                                  [(ein Drucker) 0 1]
                                  [(ein Scanner) 0 1]
                                  [(ein Joystick) 0 1]
                                  [(ein Boxen_Set) 0 1]}))
    :dokumentation "Ein Standard-PC")
```

Abbildung 3.3: Wurzelkonzept für eine PC-Konfiguration

Durch die Vererbungsstrukturen ist es insbesondere möglich, inkonsistente Konzeptbeschreibungen frühzeitig zu erkennen (vgl. Cunis 1991, S. 75).

Ein Beispiel für eine Vererbungshierarchie ist in Abbildung 3.4 auf der nächsten Seite zu sehen. Durch den Bezeichner `:oberkonzept` wird jeweils das übergeordnete Konzept in der *is-a*-Hierarchie angegeben. Das Konzept mit dem Namen `domaenenobjekt` in `PC_Komponente` ist das allgemeinste definierbare Objekt. Das Konzept `AMD_Duron_800` erbt von `AMD_Duron`, dies wiederum von `Prozessor` und dieses wiederum von dem Konzept `PC_Komponente`.

3.4.3 Kompositionelle Hierarchie

Die Taxonomie wird innerhalb der Begriffshierarchie mit der, für die Konfigurierung besonders wichtigen, *kompositionellen Hierarchie*, auch *Zerlegungshierarchie* oder *Partonomie* genannt, verknüpft. Innerhalb der kompositionellen Hierarchie wird von Aggregaten über 1–n Relationen (*has-parts*) auf ihre (Teil-)Komponenten verwiesen. Mehrfaches Vorkommen wird durch eine Stelligkeit der jeweiligen Relation definiert. In Abbildung 3.5 auf Seite 27 ist das Konzept `Festplatte` zu sehen, welches mit dem (Wurzel-)Konzept `PC`

```
(def-do
    :name PC_Komponente
    :oberkonzept domaenenobjekt
    :parameter ((Preis [0.0 inf]))
    :relationen ((komponente-von (ein PC))))

(def-do
    :name Prozessor
    :oberkonzept PC_Komponente
    :parameter ((Taktfrequenz [500 2000])
                (Typ {'Sockel_A 'Sockel_370 'Sockel_478})
                (FSB_Rate {66 100 133})
                (Icon "cpu" (non-config true))))

(def-do
    :name AMD_Duron
    :oberkonzept Prozessor
    :parameter ((Taktfrequenz {800 900 1000 1100})
                (Typ 'Sockel_A)
                (FSB_Rate 100)
                (Icon "athlon" (non-config true))))

(def-do
    :name AMD_Duron_800
    :oberkonzept AMD_Duron
    :parameter ((Taktfrequenz 800)
                (Preis 129)))
```

Abbildung 3.4: Beispiel einer Vererbungshierarchie der Taxonomie

aus Abbildung 3.3 auf der vorherigen Seite über eine *has-parts*-Relation, in diesem Fall `hat-komponente`, in Beziehung steht. Die Stelligkeit der Relation ist mit `1 2` angegeben, was bedeutet, dass ein `PC` ein oder zwei Festplatten besitzt. Insgesamt darf ein `PC` aus 7 bis 16 Konzepten vom Typ `PC_Komponente` und aus 3 bis 8 Konzepten vom Typ `Peripherie_Komponente` bestehen (siehe Abbildung 3.3 auf der vorherigen Seite).

Die *has-parts*-Beziehungen sind in der Begriffshierarchie frei definierbar. Zudem lassen sich beide Richtungen, neben *has-parts* auch dessen Inverses *part-of*, angeben. In dem Zielkonzept in Abbildung 3.3 auf der vorherigen Seite finden die *has-parts*-Beziehungen `hat-komponente` und `hat-peripherie` Verwendung. In Abbildung 3.6 auf der gegenüberliegenden Seite ist die Definition von `hat-peripherie` zu sehen. Neben dem Namen und der Klasse der Relation, lässt sich dort die inverse Relation angeben, in diesem Fall `peripherie-von`,[9] sowie die Transitivität der Relation definieren (`t` für `true`) oder ne-

[9] Außer der `has-parts-relation` und der inversen `part-of-relation` sind derzeit keine weiteren definierbaren Relationen vorgesehen. Erweiterungen sind jedoch möglich.

```
(def-do
    :name Festplatte
    :oberkonzept PC_Komponente
    :parameter ((Kapazität [20 80])
                (Geschwindigkeit {5400 7200})
                (Zugriffszeit [8.5 12])
                (Cache {512 1024 2048})
                (Icon "hd" (non-config true))))
```

Abbildung 3.5: Konzept für eine Festplatte

```
(def-relation
  :name hat-peripherie
  :klasse has-parts-relation
  :inverse peripherie-von
  :transitive t)

(def-relation
  :name peripherie-von
  :klasse part-of-relation
  :inverse hat-peripherie
  :transitive t)
```

Abbildung 3.6: Definition einer *has-parts*-Relation

gieren (`f` für `false`). Transitivität bedeutet in diesem Fall, dass Konzepte, die einem Oberkonzept zugeordnet sind, ebenfalls dessen Oberkonzept zugeordnet sind.

Mit der 1–n *has-parts*-Relation in der kompositionellen Hierarchie wird für die Konfigurierung ein *Top-Down*-Vorgehen ermöglicht: eine *Zerlegung* des Zielkonzepts in seine Dekompositionen, wobei gleichzeitig durch die Spezialisierungen in der Taxonomie eine Verfeinerung der Aggregate bis zu den Blattkonzepten erfolgt. Durch die *part-of*-Relation hingegen wird ein *Bottom-Up*-Vorgehen unterstützt, bei dem eine Aggregation von Komponenten mit dem Ziel des Zusammenfügens zu einer Integration betrieben wird. Auch wenn die Begriffshierarchie dieses ermöglichen würde, ist ein *Bottom-Up*-Vorgehen derzeit allerdings in EngCon nicht implementiert.

3.4.4 Dynamische Spezialisierung durch taxonomische Inferenzen

Innerhalb von EngCon wird zwischen zwei Arten von Komponenten unterschieden. Die *Konzepte* sind statisch in der Wissensbasis definiert, und werden zur Laufzeit nicht verändert. Sie beschreiben eine Klasse von Objekten, von denen im Laufe einer Konfigurierung beliebig viele *Instanzen* erzeugt werden können. Diese Instanzen repräsentieren zu Beginn der Konfigurierung i. d. R. nichtterminale Konzepte, d. h. Konzepte, die noch keine Blätter in der taxonomischen Hierarchie sind. Instanzen, die noch nicht endgültig spezialisiert sind, werden auch *dynamische Instanzen* genannt. Während des Konfigurierungsvorgangs werden diese Instanzen modifiziert, d. h. dynamisch spezialisiert, bis sie ein entsprechendes Blattkonzept erreicht haben. Die Spezialisierung einer Instanz kann entweder aufgrund einer interaktiv ermittelten Benutzerentscheidung oder automatisch durch taxonomische Inferenzen erfolgen.

EngCon besitzt zwei taxonomische Inferenzmodule: ein statisches und ein dynamisches. Das statische Inferenzmodul dient zur frühzeitigen Erkennung und Behebung von taxonomischen Konfliktsituationen. Dazu werden die Wertebereiche der Parameter der Oberkonzepte zur Laufzeit so weit eingeschränkt, dass der Benutzer nur noch gültige Werte der Blattkonzepte auswählen kann. Das dynamische Inferenzmodul dient dazu, aufgrund von taxonomischen Inferenzen automatische Spezialisierungen vorzunehmen. Es wird zwischen fünf Arten taxonomischer Inferenzen unterschieden (vgl. Ranze et al. 2002, S. 849):

1. automatische Spezialisierung zur nächsten Ebene,
2. automatische Spezialisierung zu einem Blattkonzept,
3. automatische Aggregat-Spezialisierung,
4. automatische Komponenten-Spezialisierung,
5. automatische Vorzerlegung.

Beispiel 3.4.1 *Es soll ein PC konfiguriert werden. Im Wurzelkonzept ist definiert, aus welchen Komponenten der PC besteht. Durch das Instantiieren des Wurzelkonzepts werden, wenn ein Minimum definiert ist (z. B.* `1 inf`*), automatisch auch die mindestens benötigten Komponenten der has-parts Relation instantiiert* ***Vorzerlegung)****.*

Wird dem Rechner vom Benutzer ein SCSI-Controller zugewiesen, wird der `PC` *automatisch zu einem* `SCSI-PC` *spezialisiert* ***(Komponenten-Spezialisierung)****. Diese Spezialisierung kann sich, in Abhängigkeit von der Definition in der Wissensbasis, auf den weiteren Konfigurierungsverlauf in Form von Einschränkungen oder Erweiterungen auf andere Eigenschaften auswirken. Die Wissensbasis könnte vorsehen, dass in einem* `SCSI-PC` *ausschließlich SCSI-Komponenten (Festplatten, CD-ROMs, etc.) verwendet werden. Die Instanz einer Festplatte würde durch die obige Entscheidung automatisch zu einer SCSI-Festplatte spezialisiert werden* ***(Aggregat-Spezialisierung)****.*

Wenn im weiteren Verlauf der Konfigurierung durch den Benutzer angegeben wird, dass eine Festplatte mit z. B. 15000 Umdrehungen/Min. gewünscht wird, so wird die Instanz von `Festplatte` *automatisch zum nächsten Unterkonzept spezialisiert, welches diese Art von Festplatten zusammenfasst* ***(Spezialisierung zur nächsten Ebene)****. Sollte in der Wissensbasis lediglich ein Konzept mit diesem Parameter-Wert vorhanden sein, wird die Instanz von* `Festplatte` *automatisch zu diesem Blattkonzept spezialisiert* ***(Spezialisierung zu einem Blattkonzept)****.*

Der Kontrollmechanismus sieht vor, dass die Konzepte sämtlicher Instanzen terminal sein müssen, d. h. sie haben keine Nachfolger zu denen sie noch spezialisiert werden könnten, damit die Konfigurierung erfolgreich abgeschlossen ist (*Top-Down*-Vorgehen). Die dazwischen liegenden Konfigurierungsschritte werden *Teilkonfigurationen* genannt.

3.4.5 Eigenschaften von Komponenten

Die Eigenschaften bzw. Attribute einer Komponente können entweder Relationen zu anderen Komponenten (wie die beschriebenen *has-parts*-Relationen) oder Parameter in Form von *Deskriptoren* sein. Ein Deskriptor ist ein Container mit einem Namen und einem Wertebereich. Dieser Wertebereich kann ein Integer, eine Fließkommazahl, ein Intervall oder eine Auswahlmenge auf Integer bzw. Fließkommazahlen, eine Zeichenkette oder eine Auswahlmenge auf Zeichenketten sein.[10]

Außerdem ist es möglich, Metainformationen zur Unterstützung des Konfigurierungsprozesses in der Wissensbasis abzulegen. Diese Metadaten werden in den Deskriptoren zugeordneten *Facetten* gespeichert. Facetten beschreiben Eigenschaften und zusätzliche Angaben von Attributen. Dies können z. B. Default-Werte, Maßeinheiten oder die Definition von Icons sein. In den Beispielen wird das Attribut `Icon` dazu verwendet, um dem Benutzer während der Konfigurierung ein Bild der aktuell zu konfigurierenden Komponente anzeigen zu können.

Innerhalb der Vererbungsstruktur können Spezialisierungen die Deskriptor-Definitionen der Oberkonzepte einschränken sowie weitere Eigenschaften hinzufügen. Nicht erlaubt hingegen ist es, Eigenschaften zu überschreiben (*strikte* Vererbung). Ein Beispiel für die Einschränkung eines Wertebereichs ist in Abbildung 3.4 auf Seite 26 zu sehen. Im Konzept `Prozessor` liegt der Wertebereich für den Parameter `Taktfrequenz` noch bei `[500 2000]`. Bei der Spezialisierung `AMD_Duron` ist das Integer-Intervall durch eine Auswahlmenge eingeschränkt worden: `{800 900 1000 1100}`.

3.5 Kontrollwissen

EngCon ist ein Expertensystem, dass Anwendung in unterschiedlichen Domänen und Systemen finden kann. Weil die Anwendungsdomänen u. U. sehr groß sein können, müssen häufig heuristische Entscheidungen getroffen werden. Mit einfachen Suchstrategien ist dies i. A. nicht effizient durchzuführen, weshalb ein flexibler Kontrollmechanismus erforderlich wird.[11] Der Kontrollmechanismus muss in der Lage sein, unterschiedliche Problemlösungsstrategien zu unterstützen und innerhalb einer Problemlösungsstrategie Möglichkeiten zur variablen, situationsabhängigen Steuerung bieten (vgl. Günter 1991a, S. 109).

Wie bereits angesprochen existiert in EngCon im Gegensatz zu „klassischen“, regelbasierten Expertensystemen wie XCon eine klare Trennung zwischen dem Domänenwissen und dem Kontrollwissen (vgl. Abschnitt 2.2.1, S. 11 und Abschnitt 3.4, S. 22). Die Ab-

[10] Intern werden auch komplexe Wertebereiche in Form von Deskriptoren genutzt, da sämtliche Komponenten der Begriffshierarchie durch unterschiedliche Deskriptoren repräsentiert werden.

[11] „Kontrolle“ nicht im Sinne von „überprüfen“, sondern von engl. *to control*: „steuern“ bzw. „lenken“. Günter (1992, S. 1) versteht unter Kontrolle „die Steuerung der Suche im Lösungsraum“.

laufkontrolle in EngCon wird von der Wissensrepräsentation implizit unterstützt, ohne dass die Begriffshierarchie selbst Kontrollwissen enthält. Das *modellbasierte* Vorgehen der Kontrolle anhand der Struktur des Domänenmodells ist ein Vorgang, bei dem die Begriffshierarchie durch die Kontrolle interpretiert wird (vgl. Günter 1991a, S. 108). Der Kontrollmechanismus von EngCon wird deshalb auch *begriffshierarchie-orientierte Kontrolle* genannt.

Die Trennung zwischen Domänen- und Kontrollwissen impliziert auch eine strikte Unterscheidung zwischen Kontroll- und Konfigurierungsentscheidungen. Konfigurierungsentscheidungen beziehen sich auf die Konfiguration an sich, also z. B. die Wertebelegung der Parameter einer Konzeptinstanz. Kontrollentscheidungen werden getroffen, um die Reihenfolge der Konfigurierungsschritte zu ermitteln. Die Aufgabe der Kontrolle ist es, hiermit die Suche im Lösungsraum zu steuern, der durch die Begriffshierarchie definiert wird. Nach jedem Konfigurierungsschritt erfolgt eine Überprüfung anhand der Begriffshierarchie, welche Schritte als nächstes jeweils noch erlaubt sind, d. h. welche Schritte zu einer konsistenten Teilkonfiguration führen. Das Ziel ist es, den Lösungsraum soweit wie möglich einzuschränken, so dass am Ende nur eine einzige konsistente Lösung übrig bleibt.

3.5.1 Kategorien von Kontrollwissen

Das Kontrollwissen in EngCon wird „explizit, deklarativ und separat von den Objektbeschreibungen repräsentiert“ (Günter 1991a, S. 108). Es wird zwischen mehreren Arten des für die Konfigurierung benötigten Kontrollwissens unterschieden.

Zur Konfigurierung in EngCon wird der Konfigurierungsvorgang in unterschiedliche Phasen eingeteilt. So kann z. B. in der ersten Phase eine „Grobkonfigurierung“ vorgenommen und ausschließlich die benötigten Komponenten ermittelt werden, um daraufhin in einer weiteren Phase die Wertebelegungen der Parameter dieser Komponenten zu bestimmen. Das für die jeweilige Phase zu verwendende Kontrollwissen wird in *Strategien* definiert (siehe Abbildung 3.7 auf der gegenüberliegenden Seite). In ihnen ist das folgende Kontrollwissen enthalten:

- **Reihenfolgewissen** definiert Auswahlkriterien, durch die der jeweils nächste Konfigurierungsschritt ermittelt wird.
- **Bearbeitungswissen** definiert Bearbeitungsverfahren, die in der jeweiligen Phase zur Ermittlung von Werten benutzt werden.
- **Fokussierungswissen** wird genutzt, um aus Effizienzgründen den Suchraum um die Elemente zu verkleinern, die für die aktuelle Problemlösung nicht benötigt werden.

Jede Strategie besitzt darüber hinaus eine Priorität, mit der die Reihenfolge zur Abarbeitung der Phasen festgelegt wird.

Die Ablaufkontrolle von EngCon ist *agenda-basiert*, d. h. dass alle in der aktuellen Phase möglichen Konfigurierungsschritte in einer *Agenda* zusammengefasst und aufgelistet werden. Die Einträge dieser Agenda werden mit Hilfe von *Agenda-Auswahlkriterien* (Reihenfolgewissen) sortiert, so dass der gewünschte Konfigurierungsverlauf entsteht. Zur

```
(def-strategie
    :name                     rechner-typ-festlegen
    :strategie-klasse         tiefensuche-strategie
    :ks-fokus                 pc-konfigurierung-fokus
    :agenda-auswahlkriterien (bevorzuge-pc-zerlege-komponenten
                              bevorzuge-pc-zerlege-peripherie)
    :bearbeitungsverfahren   (azw
                              berechnungsfkt
                              defaultuebernahme
                              dynamischer-default
                              benutzeranfrage)
    :prioritaet              100)
```

Abbildung 3.7: Beispiel für eine Strategie

Steigerung der Effizienz wird jeweils mit Hilfe eines *Konfigurierungsschritt-Fokus* (KS-Fokus) die Generierung der Agenda auf die von der jeweiligen Phase benötigten Teile beschränkt, so dass nur noch die entsprechend benötigten Konfigurierungsschritte in die Agenda eingetragen werden (Fokussierungswissen).

3.5.2 Konfigurierungsvorgang

Die Reihenfolge des Ablaufs einer Konfigurierung ist zum größten Teil frei gestaltbar. Sie ist abhängig von der Art der Aufgabe, der gewählten Strategie und dem Ausmaß der Benutzerinteraktion. Für unterschiedliche Phasen können unterschiedliche Problemlösungsstrategien vorgeben werden, die je nach Kenntnisgrad des Benutzers automatisch eine möglichst hohe Qualität der Konfiguration durch Automatisierung (z. B. mittels Defaults) erzeugen oder eine hohe Interaktivität des Benutzer erfordern (Eingabe vieler Parameter, Spezialisierungen oder Zerlegungen). So ist bspw. ein Benutzer „Experte“ für Mainboards und möchte die Konfigurierung an dieser Stelle mit konkreten Werten selbst beeinflussen. Gleichzeitig ist er allerdings z. B. Grafikkarten betreffend nur „Laie“ und sollte entsprechend durch das System unterstützt werden.

Der Ablauf der Konfigurierung orientiert sich an der Begriffshierarchie, in der in einem Metamodell entsprechend der CWA die Menge aller zulässigen Konfigurierungen deklarativ definiert sind, um eine vollständige, „widerspruchsfreie Instantiierung der Begriffshierarchie zu finden, die den Anforderungen der Aufgabenstellung genügt“ (Günter 1992, S. 117).[12] Während des Konfigurierungsverlaufs werden Instanzen von den Konzepten der Begriffshierarchie erzeugt, schrittweise spezialisiert und zu einer Konfiguration zusammengefügt. Dabei wird die jeweils aktuelle Teilkonfiguration mit der Begriffshierarchie verglichen, um festzustellen, welche Konfigurierungsschritte als nächstes „syntaktisch“ durchführbar sind (vgl. Günter 1991a, S. 97). Es wird zwischen drei *Konfigurierungsschritt-Typen* (KS-Typen) unterschieden:

[12] „Vollständig“ in diesem Fall bezogen auf die Begriffshierarchie.

- **Spezialisierung:** Eine dynamische Instanz wird entsprechend der Definition in der Begriffshierarchie zu einer Instanz des spezielleren Konzeptes verfeinert.
- **Parametrierung:** Die Wertebereiche von Parametern (Slots) von Instanzen werden entweder festgelegt oder weiter eingeschränkt.
- **Zerlegung:** Eine Konfigurierungsinstanz wird entsprechend der kompositionellen Hierarchie in seine Komponenten zerlegt. Dies impliziert ein *Top-Down*-Vorgehen.

Die Vorgehensweise des Kontrollmechanismus ist „opportunistisch", d. h. es wird mittels Agenda-Auswahlkriterien bestimmt, welcher nächste Konfigurierungsschritt laut Spezifikation des Wissensingenieurs am erfolgversprechendsten ist und dieser entsprechend durchgeführt (vgl. Cunis und Günter 1991, S. 39). Agenda-Auswahlkriterien sind „Muster", die mit den Konfigurierungsschritten der Agenda verglichen werden. Der Konfigurierungsschritt, der am Besten die Vorgaben durch die Auswahlkriterien erfüllt (d. h. die meisten Übereinstimmungen ausweist), wird jeweils als nächstes zur Durchführung ausgewählt.

Der Konfigurierungsverlauf in EngCon kann vereinfacht als Zyklus betrachtet werden. Es werden der Reihe nach Konfigurierungsschritte aus der Agenda ausgewählt und abgearbeitet. Der vereinfachte, elementare Teil des *zentralen Zyklus* des Kontrollmechanismus von EngCon umfasst die folgenden Schritte (vgl. Günter 1991a, S. 97):

1. Analyse der aktuellen Teilkonfiguration zur Bestimmung syntaktisch möglicher Konfigurierungsschritte und Aufnahme der Schritte in die Agenda.
2. Mittels Agenda-Auswahlkriterien wird ein geeigneter Konfigurierungsschritt aus der Agenda ausgewählt.
3. Der ausgewählte Konfigurierungsschritt wird mit einem entsprechenden Bearbeitungsverfahren durchgeführt.
4. Propagation des Constraint-Netzes, um die aktuelle Teilkonfiguration konsistent zu halten bzw. Inkonsistenzen feststellen zu können.

Der Zyklus ist danach abgeschlossen und der Kontrollmechanismus fährt mit der Strategie mit der nächsthöchsten Priorität fort. Die Konfigurierung ist erfolgreich beendet, wenn alle Strategien abgearbeitet, alle Konzeptinstanzen vollständig zu Blattkonzepten spezialisiert wurden und sich die Teilkonfiguration in einem konsistenten Zustand bezüglich der Begriffshierarchie befindet.

Eine detaillierte Beschreibung des vollständigen Konfigurierungszyklus von EngCon, einschließlich Beispielen und Erläuterungen zum KS-Fokus, Agenda-Auswahlkriterien und Bearbeitungsverfahren befindet sich in Anhang B auf Seite 272 ff.

3.5.3 Interaktives Konfigurieren

Für den Konfigurierungsverlauf mit EngCon ist kennzeichnend, dass der Benutzer weitestgehend Einfluss auf den Ablauf der Konfigurierung nehmen kann. Interaktiv wird mit dem Benutzer eine für ihn geeignete Lösung gesucht. Es bestehen konkret die folgenden Interaktionsmöglichkeiten (vgl. Günter 1992, S. 172):

- Beim Start der Konfigurierung muss vom Benutzer das Zielkonzept und damit das Konfigurierungsziel vorgegeben werden.
- Der Benutzer kann jederzeit die aktuell zu verwendende Strategie ändern.
- Aus der generierten Agenda kann der Benutzer einen zu bearbeitenden Konfigurierungsschritt auswählen, anstatt den vorgegebenen Schritt auszuführen.
- Werte, für die das Bearbeitungsverfahren *Benutzereingabe* definiert ist, werden interaktiv durch den Benutzer festgelegt. Ebenso können alle Wertebelegungen nachträglich geändert werden.[13]
- Konflikte können interaktiv mit dem Benutzer gelöst werden.
- Die gesamte Konfigurierung kann jederzeit durch den Benutzer abgebrochen werden.

Innerhalb der Strategien legt der Wissensingenieur fest, welche Interaktionsmöglichkeiten der Benutzer während der Konfigurierung haben soll bzw. welche notwendig sind. Mit den zur Verfügung stehenden Mitteln ist es möglich, das Spektrum von vollautomatischen bis zu hochgradig interaktiven Systemen zu realisieren.

3.6 Constraint-Wissen

Constraints werden in Konfigurierungssystemen zur Repräsentation und Verarbeitung von Konfigurierungsrestriktionen eingesetzt. Da diese Restriktionen u. U. einen großen Teil des statischen Wissens ausmachen können, sollte das Constraint-System eine adäquate Modellierungssprache und effiziente Inferenz- und Konsistenzmechanismen aufweisen. Im Detail dienen Constraints in Konfigurierungssystemen

- zur Beschreibung von Abhängigkeiten zwischen Konfigurierungsobjekten,
- zur Konsistenzüberprüfung, ob die aktuelle Teilkonfiguration den Restriktionen genügt,
- zur Einschränkung der Eigenschaften von Konfigurierungsobjekten entsprechend den Restriktionen,
- zur Berechnung von Eigenschaftswerten von Konfigurierungsobjekten, für die noch kein Wertebereich ermittelt wurde, anhand bestehender Eigenschaftswerte anderer Konfigurierungsobjekte (vgl. Günter 1992, S. 73).

[13] Sofern für die betreffenden Wertebelegungen nicht das Bearbeitungsverfahren AZW (*allgemein zulässiger Wertebereich*) vorgesehen wurde (vgl. Anhang B, S. 272, Schritt 7).

In EngCon werden durch Constraints formulierte Beschränkungen dazu genutzt, Abhängigkeiten zwischen den Parametern der Konfigurierungsobjekte zu beschreiben (vgl. Ranze et al. 2002, S. 850). Mit Constraints ist es möglich, komplexe Abhängigkeiten, wie funktionale Zusammenhänge und Restriktionen zwischen Objekten der Begriffshierarchie, auf Konzeptebene deklarativ zu beschreiben. Dabei sind sie keine Mittel der objektorientierten Wissensrepräsentation, denn sie werden i. d. R. auf mehrere Objekte angewandt und es ist keine eindeutige Zuordnung zu einem bestimmten Objekt möglich.

Das Constraint-Modell von EngCon basiert auf den von Gülden (1993) beschriebenen Konzepten zur Integration eines Constraint-Systems in KonWerk, dem Vorgänger von EngCon. Es lässt sich in drei Komponenten einteilen, die in den folgenden Abschnitten detailierter beschrieben werden:[14]

1. **Konzeptuelle Constraints** beschreiben bestehende Abhängigkeiten zwischen den Konzepten der Domäne. Sie werden genutzt, um durch sie eine Zuordnung zwischen den Attributen der Konfigurierungsobjekte und den Variablen der Constraint-Relationen vornehmen zu können.

2. **Constraint-Relationen** beschreiben die konkreten Abhängigkeiten zwischen Constraint-Variablen. Über konzeptuelle Constraints werden die Variablen der Relationen an Attribute der Konfigurierungsobjekte gebunden. Constraint-Relationen werden bei Bedarf instantiiert und stehen über ein Constraint-Netz zueinander in Verbindung.

3. Das **Constraint-Netz** repräsentiert alle durch Constraints definierten Abhängigkeiten der aktuellen Teilkonfiguration. Im Constraint-Netz werden die instantiierten Constraint-Relationen und deren Abhängigkeiten untereinander verwaltet. Abhängigkeiten zwischen Constraint-Relationen treten auf, wenn sich mehrere Relationen auf dieselben Parameter von Konfigurierungsobjekten beziehen, d. h. wenn an den Slot eines Konfigurierungsobjektes mehrere Constraint-Variablen von instantiierten Constraint-Relationen gebunden sind.

Ein wichtiger Aspekt ist, dass das Constraint-Netz während des Konfigurierungsvorgangs inkrementell anwächst. Nach und nach, immer wenn neue Komponenten aus der Begriffshierarchie dynamisch während der Konfigurierung erzeugt werden, zwischen denen Abhängigkeiten über konzeptuelle Constraints definiert sind, werden entsprechend neue Constraint-Relationen instantiiert und dem Constraint-Netz hinzugefügt.

Die Interaktion des Constraint-Systems mit EngCon erfolgt über eine relativ schlanke Schnittstelle. Vor einer Propagation der Constraints werden die Constraint-Variablen mit den Wertebereichen aus der aktuellen Teilkonfiguration belegt bzw. aktualisiert. Die internen Mechanismen zur Auswertung innerhalb des Constraint-Systems laufen wie in einer *Black Box* unabhängig vom restlichen System ab und lassen sich von der Kontrollkomponente nach dem Anstoßen der Propagation bis zu deren Ende nicht weiter steuern.

[14] Das domänenunabhängige Constraint-System von EngCon bzw. KonWerk geht ursprünglich auf das von Güsgen (1989) bei der „Gesellschaft für Mathematik und Datenverarbeitung“ (GMD) entwickelte ConSat-System zurück (vgl. Cunis und Günter 1991, S. 46).

```
(def-konzeptuelles-constraint
    :name                    conc_AGP_Mainboard
    :variablen-pattern-paare ((?v :name VGA_Card
                                  :parameter((Bus 'agp)))
                              (?m :name Mainboard))
    :constraint-aufrufe      ((func_AGP_Mainboard (?m AGP_Slot))))
```

Abbildung 3.8: Konzeptuelles Constraint

Am Ende einer Propagation werden die aktualisierten und neu berechneten Wertebereiche wiederum in die aktuelle Teilkonfiguration übertragen.

3.6.1 Konzeptuelle Constraints

Konzeptuelle Constraints sind neben dem in der Begriffshierarchie definierten Konzept- und Kontrollwissen Teil des statischen Wissens. Sie beschreiben Abhängigkeiten zwischen den Konzepten der Begriffshierarchie von EngCon. Mittels konzeptueller Constraints lässt sich über die Konzeptebene eine Zuordnung zwischen den Instanzen bestimmter Konzepte und deren Parametern zu anderen Instanzen vornehmen. In konzeptuellen Constraints werden Instantiierungsregeln, auch „Bindungsmuster" genannt, in Form von sog. *Variablen-Pattern-Paaren* definiert. Mit einem Variablen-Pattern-Paar wird eine bestimmte Instanz eines Konfigurierungsobjektes an eine *Pattern-Variable* gebunden. Variablen-Pattern-Paare werden von einem *Pattern-Matcher* ausgewertet. Der Pattern-Matcher entscheidet, wann die in den konzeptuellen Constraints definierten Constraint-Relationen instantiiert werden. Für jede neu erzeugte Instanz eines Konfigurierungsobjektes überprüft der Pattern-Matcher, ob mit ihr das Variablen-Pattern-Paar eines konzeptuellen Constraints erfüllt wird (engl. *to match*). Ist ein Variablen-Pattern-Paar erfüllt, werden Instanzen von der bzw. den in dem konzeptuellen Constraint definierten Constraint-Relation(en) erzeugt und deren Constraint-Variablen an die Slots der Instanzen der betreffenden Konfigurierungsobjekte gebunden (vgl. Werres 2002).

Beispiel 3.6.1 *Das konzeptuelle Constraint mit dem Namen `conc_AGP_Mainboard` in Abbildung 3.8 bestimmt, dass wenn die Variablen-Pattern-Paare erfüllt sind, die Constraint-Relation `func_AGP_Mainboard` mit dem Parameter `AGP_Slot` instantiiert wird. Die Instantiierungsregeln bzw. die Variablen-Pattern-Paare, in denen Objektinstanzen an die Pattern-Variablen `?v` und `?m` gebunden werden, besagen, dass genau dann eine Instanz der Constraint-Relation erzeugt wird, wenn es*

1. *eine Instanz des Konzepts `VGA_Card` mit der Wertebelegung `agp` für den Parameter `Bus` und*

2. *eine Instanz des Konzepts mit dem Namen `Mainboard` gibt (deren Parameter `AGP_Slot` der Instanz der Constraint-Relation übergeben wird).*

Über konzeptuelle Constraints wird somit durch sukzessives Instantiieren von Constraints der inkrementelle Aufbau des Constraint-Netzes gesteuert. Konzeptuelle Constraints können wie (Meta-)Regeln mit einem *Bedingungs-* und einem *Aktionsteil* aufgefasst werden. Der Bedingungsteil wird dementsprechend durch die Variablen-Pattern-Paare repräsentiert, der Aktionsteil durch den Constraint-Aufruf bzw. die Instantiierung der Constraint-Relation.

3.6.2 Constraint-Relationen

In EngCon werden abstrakte, domänenunabhängige Constraints durch *Constraint-Relationen* beschrieben. Sie werden innerhalb der konzeptuellen Constraints definiert und sind Teil des Constraint-Wissens der Wissensbasis. Auch Constraint-Relationen haben per se keinen Bezug zu konkreten Konfigurierungsobjekten. Der Bezug wird erst durch die Zuordnung des Pattern-Matchers vorgenommen (vgl. Abschnitt 3.6.1 auf der vorherigen Seite). Es wird zwischen drei unterschiedlichen Klassen für Constraint-Relationen unterschieden (vgl. Ranze et al. 2002, S. 850):

1. **Funktions- und Prädikat-Constraints**
   Komplexe, funktionale Zusammenhänge können als Gleichungen formalisiert werden. Durch sie lässt sich z. B. physikalisches und mechanisches (Abhängigkeits-)Wissen beschreiben, dass je nach Anwendung einen wesentlichen Teil des Expertenwissens ausmachen kann. Funktions- und Prädikat-Constraints beinhalten Gleichungen und Ungleichungen ($=, \leq, \geq, <, >$) zur Beschreibung von Wertezuweisungen sowie Verhältnisabhängigkeiten und werden von EngCon mit Hilfe eines externen Constraint-Solvers propagiert.

2. **Extensionale Constraints/Tupel-Constraints**
   Extensionale Constraints (Attribut-Wert-Tupel) sind relationale Abhängigkeiten, die als Aufzählung von Tupeln aller zulässigen Wertekombinationen der Constraint-Variablen in einer Tabelle abgelegt und von EngCon dynamisch mit Hilfe einer externen Datenbank und entsprechenden Abfragemöglichkeiten ausgewertet werden.

3. **Java-Constraints**
   Keine Constraints im ursprünglichen Sinne sind die Java-Constraints von EngCon, da sie nicht deklarativ sind und nicht per se eine bidirektionale Auswertung garantieren. Java-Constraints sind vielmehr ein flexibler Mechanismus, mit dem Funktionen und Berechnungen ausgeführt werden können, die sich mit den anderen Constraint-Arten nicht oder nur sehr umständlich realisieren lassen würden. Sie ergänzen die Constraint-Funktionalität von EngCon durch die Möglichkeit der Implementierung von prozeduralen Constraints (ähnlich einer Berechnungsfunktion) auf der Ebene von Programm-Code in Form von Java-Methoden.

Die Definition eines Constraints sollte nach Möglichkeit so abstrakt wie möglich, d. h. als Funktions- bzw. Prädikat-Constraint, erfolgen. In Szenarien mit überschaubaren Lösungsmengen bietet es sich allerdings zweckmäßigerweise an, Constraints zu nutzen, die

```
(def-constraint-relation
    :name                     func_CD_Rom
    :constraint-typ           :funktion
    :externe-pins             (A B)
    :beschraenkungs-funktion  "A = 150 * B")

(def-constraint-relation
    :name                     tpc_FSB_Rate
    :constraint-typ           :tupel
    :externe-pins             (MB_FSB_Rate P_FSB_Rate S_FSB_Rate)
    :database                 "PC_DB")

(def-constraint-relation
    :name                     jc_Price
    :constraint-typ           :java
    :externe-pins             (A B C)
    :classpath                "."
    :class                    "PC_AdvancedLib"
    :method                   "partialPriceForPC"
    :dokumentation            "Aufsummieren der Preise für A und B")
```

Abbildung 3.9: Constraint-Relationen

als Tupel von konkreten Attribut-Werten modelliert werden. Extensionale Constraints sollten entsprechend immer dann eingesetzt werden, wenn die Relation endlich ist und eine geringe Mächtigkeit aufweist, oder wenn sie sich nicht bzw. nur sehr aufwendig abstrakt spezifizieren lässt. Das Constraint kann in diesem Fall einfach durch Angabe aller zulässigen Wertekombinationen als Tupel definiert werden. Für nicht endliche oder sehr umfangreiche Relationen, d. h. für Relationen, die eine unendliche oder sehr große Lösungsmenge aufweisen, dienen Funktions- bzw. Prädikat-Constraints. Mit ihnen lassen sich Gleichungen und Ungleichungen angeben, aufgrund denen entschieden wird, welche Tupel zur Relation gehören und welche nicht, bzw. in welcher Form die Wertebereiche eingeschränkt werden müssen. Werden noch flexiblere Mechanismen benötigt, kann auf Java-Constraints zurückgegriffen werden. Sie erlauben es, beliebigen Programm-Code auf die übergebenen Attribute anzuwenden.[15] Java-Constraints können z. B. Verwendung finden bei der Aufsummierung der Attribut-Werte unterschiedlicher Konzeptinstanzen oder um Anfragen, z. B. die Verfügbarkeit von Komponenten betreffend, an externe Systeme zu stellen.

Constraint-Relationen werden mittels einer deklarativen Beschreibung in einer Meta-Modellierungssprache innerhalb der Wissensbasis definiert. Sie können dadurch relativ einfach modifiziert bzw. erweitert werden. Zur Verdeutlichung sind in Abbildung 3.9 mehrere Constraint-Relationen aufgeführt. Das Funktions-Constraint mit dem Namen

[15]Da die Korrektheit eines Java-Constraints von der jeweiligen Implementierung abhängig ist, können Java-Constraints bei intensiver Nutzung eine nicht unerhebliche, potentielle Fehlerquelle darstellen.

#	MB_FSB_Rate	P_FSB_Rate	S_FSB_Rate
1.	66	66	66
2.	66	66	100
3.	66	66	133
4.	100	100	100
5.	100	100	133
6.	133	133	133

Tabelle 3.1: Datenbank-Tabelle für ein Tupel-Constraint

`func_CD_Rom` aus berechnet beispielhaft die Datentransfer-Rate `A` für ein CD-ROM-Laufwerk anhand des Multiplikators `B` für die CD-ROM-Geschwindigkeit.[16] Das entsprechende konzeptuelle Constraint, welches den Aufruf für die Constraint-Relation `func_CD_Rom` enthält, übergibt bei der Instantiierung zwei Parameter, die den externen Pins, bzw. den Constraint-Variablen, `A` und `B` zugewiesen werden. Der in EngCon derzeit eingesetzte externe Constraint-Solver verwendet intern zur Repräsentation und Auswertung der Wertebereiche von Constraint-Variablen reellwertige Intervalle und Methoden der von Hyvönen (1992) vorgestellten „Toleranzpropagation“ (vgl. Abschnitt 5.3.4, S. 154 ff.).

Ebenfalls in Abbildung 3.9 auf der vorherigen Seite ist ein Tupel-Constraint mit dem Namen `tpc_FSB_Rate` spezifiziert. Dieses Constraint dient dazu, bei einer PC-Konfigurierung von der Taktfrequenz des *Front-Side-Bus* des Mainboards entsprechend auf die Taktfrequenzen für die CPU und den RAM-Speicher schließen zu können und umgekehrt, bzw. um selbige eingrenzen zu können. Die externen Pins `MB_FSB_Rate`, `P_FSB_Rate` und `S_FSB_Rate` stehen entsprechend für die Taktfrequenzen von *Mainboard*, *Prozessor* und *Speicher*. In Tabelle 3.1 sind die Abhängigkeiten der Taktfrequenzen dargestellt. Eine ebensolche Tabelle muss zur Nutzung und Auswertung für EngCon unter dem Namen des Constraints (`tpc_FSB_Rate`) in einer relationalen Datenbank hinterlegt werden.[17] Jede Constraint-Relation vom Typ `tupel` entspricht einer Tabelle innerhalb der Datenbank. Eine Spalte der Datenbank-Tabelle entspricht jeweils einem Pin des Constraints. Es sind ausschließlich atomare Datentypen (String, Integer, Float) erlaubt. Im Gegensatz zu Funktions- bzw. Prädikat-Constraints sind keine intervallwertigen Wertebereiche möglich. Die Datenbank muss dem System unter dem in der Constraint-Relation angegebenen (ODBC-)Namen (hier: `PC_DB`) bekannt sein, damit eine Auswertung durch EngCon erfolgen kann.

Beispiel 3.6.2 *Eine SQL-Anfrage zur Auswahl bzw. Einschränkung der Wertebereiche für die Parameter, die an die Constraint-Variablen der Tabelle 3.1 gebunden sind, könnte folgendermaßen aussehen:*

[16] 150 kB/s entsprechen 1-facher CD-ROM-Geschwindigkeit (sog. *Single-Speed*).

[17] Extensional repräsentierte Constraints müssen keinesfalls zwingend in einer Datenbank verwaltet bzw. ausgewertet werden, wie in Kapitel 5 ersichtlich wird. Aufgrund des unkonventionellen und zugleich pragmatischen Ansatzes in EngCon bzgl. der Auswertung extensionaler Constraints durch Datenbank-Abfragen, wird an dieser Stelle eine ausführliche Erläuterung der praktischen Umsetzung dieses Ansatzes gegeben.

```
SELECT * FROM tpc_FSB_Rate WHERE (MB_FSB_Rate IN (66, 100, 133)) AND
                                 (P_FSB_Rate EQ 100) AND
                                 (S_FSB_Rate IN (100, 133))
```

Die Anfrage würde als Ergebnis die Zeilen 4 und 5 aus der Tabelle 3.1 zurückgeben, aus denen entsprechend die neuen, eingeschränkten Wertebereiche der Constraint-Variablen abgeleitet werden können.

Das Java-Constraint mit dem Namen `jc_Price` in Abbildung 3.9 auf Seite 37 wird dazu genutzt, den Gesamtpreis mehrerer Komponenten zu summieren, in diesem Beispiel der beiden Instanzen, die an die Constraint-Variablen `A` und `B` gebunden sind. Der letzte externe Pin (hier: `C`) nimmt jeweils den berechneten Rückgabewert der Java-Methode auf. In diesem Fall heißt die Methode `partialPriceForPC`, die sich innerhalb der Klasse `PC_AdvancedLib` befinden muss. Der Pfad zu dem Java-Package, in dem sich die Klasse `PC_AdvancedLib` befindet, wird über `classpath` spezifiziert.

3.6.3 Constraint-Netz

Instanzen von Konfigurierungsobjekten werden während des Konfigurierungsverlaufs nach und nach durch die Kontrolle bzw. durch den Pattern-Matcher, basierend auf den Konzepten der Begriffshierarchie, instantiiert. Weil mit jeder neuen Instanz, je nach Spezifikation in der Wissensbasis, Constraint-Abhängigkeiten bzgl. der Attribute eingehalten werden müssen, werden entsprechende Instanzen der Constraint-Relationen ebenfalls dynamisch während der Konfigurierung erzeugt. Sie bilden ein inkrementell anwachsendes Constraint-Netz, dessen Abhängigkeiten erfüllt werden müssen, um eine konsistente Konfiguration zu erhalten.

Ein simples Beispiel für ein Constraint-Netz einer PC-Konfigurierung ist in Abbildung 3.10 auf der nächsten Seite zu sehen. Die Abhängigkeiten des Tupel-Constraints aus Tabelle 3.1 auf der vorherigen Seite sind hier als Funktions-Constraints spezifiziert. Die Knoten entsprechen den Constraint-Variablen, die Kanten repräsentieren die Constraints. Die Taktfrequenzen für das Mainboard und den Prozessor müssen identisch sein, während die Taktfrequenz für den Speicher jeweils größer oder gleich sein darf.

Zur Propagation des Constraint-Netzes werden unbekannte oder unterspezifizierte Werte aus bereits bekannten Slot-Werten abgeleitet bzw. die vorhandenen Wertebereiche eingeschränkt (vgl. Günter 1992, S. 100). Am Ende einer Propagation wird ein bestimmter Konsistenzgrad erreicht, was einer Problemreduktion entspricht. Ein Constraint ist konsistent, wenn für jeden Wert aus der Domäne einer Constraint-Variable konsistente Werte in den Domänen der übrigen, in dem Constraint vorkommenden Variablen existieren, so dass das Constraint erfüllt ist. Es wird allgemein zwischen *lokaler* und *globaler* Konsistenz unterschieden. Während bei globaler Konsistenz das gesamte Constraint-Netz konsistent sein muss, bedeutet lokale Konsistenz, dass jeweils nur, je nach lokalem Konsistenzgrad, bestimmte Bereiche des Constraint-Netzes konsistent sind. Ausführlich wird auf unterschiedliche Propagationsmechanismen in Kapitel 5 eingegangen.

Die Propagation des Constraint-Netzes erfolgt bei jedem durch die Kontrolle gesteuerten Konfigurierungszyklus. Die Constraint-Variablen müssen dazu vorher mit den aktuel-

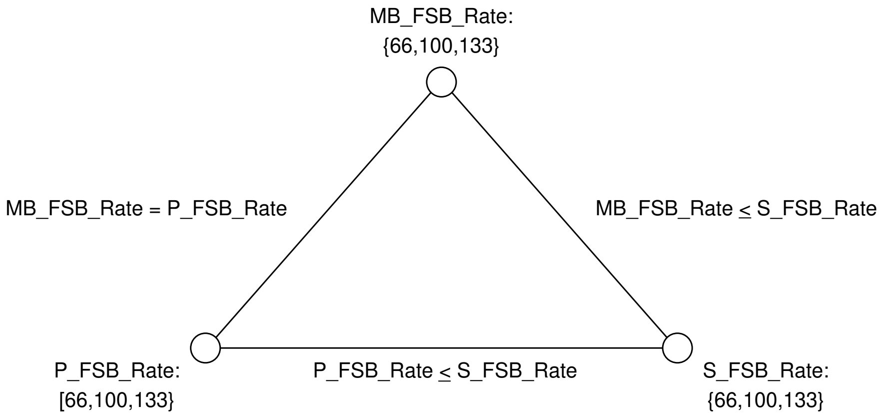


Abbildung 3.10: Beispiel für ein Constraint-Netz

len Wertebelegungen der Slots der Konfigurierungsobjekte belegt sein. Danach erfolgt die eigentliche Propagation durch das Constraint-System. Abschließend werden wiederum die propagierten Wertebereiche der Constraint-Variablen aus dem Constraint-System zurück in die entsprechenden Slots der Konfigurierungsobjekte der aktuellen Teilkonfiguration geschrieben.

3.6.4 Visualisierung des Constraint-Netzes

Die Visualisierung des Constraint-Netzes geschieht innerhalb von EngCon mit Hilfe einer Tabelle. In Abbildung 3.11 auf der gegenüberliegenden Seite ist das Constraint-Netz einer PC-Konfigurierung zu sehen. In der linken Spalte der Tabelle werden die Pins der Constraint-Relationen, die Constraint-Variablen, angezeigt. Rechts daneben erfolgt die Ausgabe der aktuellen Wertebereiche der Pins, gefolgt von den Constraints, für die durchnummeriert (`C0` bis `C9`) jeweils eine Spalte steht. Ein Kreuz bezeichnet, dass die Constraint-Variable mit dem Constraint in der jeweiligen Spalte in Relation zu anderen Variablen steht, die, wiederum durch ein Kreuz gekennzeichnet, ebenfalls zu diesem Constraint gehören. Aus der Tabelle lässt sich z. B. ablesen, dass der Parameter `FSB_Rate` der Konzeptinstanz `Memory_7` über das Constraint `C7` in Relation zum Parameter `FSB_Rate` der Instanz `Processor_3` und zum Parameter `FSB_Rate` von `Mainboard_1` steht.

3.7 Diskussion zum Constraint-System

Im Folgenden wird der Constraint-Formalismus in Bezug auf mögliche Alternativen diskutiert und die Besonderheiten des Constraint-Systems von EngCon aufgezeigt. Besondere Beachtung findet bei letzterer Betrachung der dynamische Mechanismus der konzeptuellen Constraints sowie die unterschiedlichen Constraint-Klassen von EngCon.

Pin	Value	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
PC 0.Price	**[434.0 .. 930.0]**	X									
Processor 3.Type	**Slot_A**		X								
Tower 2.power supply	**300**			X							
CD Rom 5.Speed	**24 32 40**				X						
PC 0.RAM check	**0 1 2**					X	X				
Mainboard 1.AGP Slot	**1**							X			
Memory 7.FSB Rate	**100**					X	X		X	X	
Processor 3.FSB Rate	**100**								X	X	X
CD Rom 5.Transfer Rate	**3600 4800 6000**				X						
Processor 3.Price	**[370 .. 370]**	X									
Mainboard 1.Type	**Slot_A**		X								
Memory 7.Price	**[64 .. 560]**	X									
Mainboard 1.FSB Rate	**100**								X		X

Abbildung 3.11: Visualisierung des Constraint-Netzes in EngCon

3.7.1 Constraints und mögliche Alternativen

Constraints sind im Gegensatz zu Berechnungsfunktionen und Regeln ungerichtet. Daraus ergibt sich die markanteste Eigenschaft des Constraint-Formalismus, nämlich die Möglichkeit zur multidirektionalen Auswertung von Constraints-Ausdrücken. Hierfür wird das Constraint-Netz propagiert, bis dem Lösungsverfahren entsprechende, konsistente Wertebereiche für die Constraint-Variablen ermittelt worden sind. Die Propagation eines Constraints bzw. eines Constraint-Netzes kann sich allerdings sehr aufwendig gestalten, je nachdem wie umfangreich das Constraint-Netz ist, welches Propagationsverfahren zum Einsatz kommt und was für ein Konsistenzgrad erreicht werden soll.

Alternativen zum Constraint-Formalismus, wie Berechnungsfunktionen und Regeln, sind dagegen gerichtet, d. h. im Gegensatz zu Constraints erfolgt die Auswertung ausschließlich unidirektional. Es kann immer nur ein vorher festgelegter Wert berechnet werden. Auch wenn dieser bereits gegeben ist, lassen sich aufgrund dessen keine weiteren Wertebelegungen ableiten. Allerdings sind Berechnungsfunktionen und Regeln dadurch effizienter als Constraints. Sie werden nur einmal ausgeführt, und im Gegensatz zu Constraints können Berechnungsfunktionen als compilierter Code vorliegen (vgl. Syska und Cunis 1991, S. 89 ff.; Günter 1992, S. 152).

Wenn sich die Wertebereiche von Constraint-Variablen geändert haben, werden die zugehörigen Constraints neu berechnet, bzw. das Constraint-Netz wird so lange propagiert, bis wieder ein konsistenter Zustand erreicht wird. Im Gegensatz dazu „feuern" Regeln nur ein einziges mal, nämlich wenn ihr Bedingungsteil erfüllt ist. Auch Berechnungsfunktionen werden nur einmal, festgelegt durch die Kontrolle, ausgeführt. In diesem Fall muss die Kontrolle den Zeitpunkt bestimmen und sicherstellen, dass alle zur Berechnung benötigten Variablen mit bekannten Wertebereichen belegt sind. Es wird hierbei durch die Kontrolle eine Reihenfolge ermittelt, in der die Berechnungsfunktionen ausgeführt werden können, zwischen denen u. U. mehrfach verkettete Abhängigkeiten bestehen. Randbedingungen werden dadurch, wie beim Regelmechanismus, in das Kontrollwissen verlagert.

Constraints dagegen sind ein deklarativer Formalismus, d. h. sie sind frei von Kontrollstrukturen. Die Reihenfolge der Auswertung ist nicht festgelegt und für das Ergebnis nicht relevant. Der Wissensingenieur spezifiziert die Abhängigkeiten und überlässt die Auswertung vollständig dem System. Das Constraint-System kann daher als domänenunabhängiges, geschlossenes Modul, als sog. *Black Box*, betrachtet werden (vgl. Günter 1992, S. 75; Syska und Cunis 1991, S. 78). Dies erleichtert zudem die Konflikterkennung: Bei der Propagation durch das Constraint-Modul werden bei jedem Durchlauf des Propagationsalgorithmus sämtliche Abhängigkeiten überprüft und Konflikte bzw. Inkonsistenzen im Constraint-Netz sofort erkannt. Bei Berechnungsfunktionen hingegen ist der Zeitpunkt der Ausführung abhängig von den Eingangsvariablen und der vorher festgelegten Ausführungsreihenfolge. Ein Konflikt wird u. U. erst sehr spät bei der Auswertung der jeweiligen Funktion erkannt. Wird ein nicht zulässiger Wert berechnet, müssen ggf. vorhergehende Konfigurierungsschritte zurückgenommen werden (vgl. Günter 1992, S. 152; Syska und Cunis 1991, S. 90 f.).

3.7.2 Besonderheiten des Constraint-Systems von EngCon

Neben diesen allgemeinen Eigenschaften von Constraints gibt es einige spezifische Besonderheiten des Constraint-Systems von EngCon. So ist die leistungsfähige Constraint-Komponente von EngCon im Gegensatz zu klassischen (statischen) Constraint-Systemen ein *dynamisches* System mit *generischen* Constraints. Durch Bindungsmuster in Form von konzeptionellen Constraints wird während des Konfigurierungsverlaufs dynamisch ein inkrementell anwachsendes Constraint-Netz bestehend aus Instanzen von Constraint-Relationen instantiiert. Da die in der Wissensbasis definierten Constraint-Relationen nicht an konkrete Konfigurierungsobjekte gebunden sind, sondern sich auf Konzepte, d. h. Typen von Komponenten, beziehen, können sie als generische Constraints in einem Metamodell gesehen werden, die durch konzeptionelle Constraints an konkrete Instanzen gebunden werden.

Bezogen auf die einzelnen, EngCon-spezifischen Constraint-Typen ergeben sich jeweils unterschiedliche Vor- und Nachteile. So sind die Java-Constraints in EngCon zwar sehr flexibel, allerdings müssen die Auswertemechanismen für jedes Java-Constraint vom Wissensingenieur auf Quellcode-Ebene implementiert werden. Dies setzt tiefergehende Programmierkenntnisse voraus und kann sich, je nach Komplexität der zu implementierenden Abhängigkeiten, in der Praxis als ein fehleranfälliges Verfahren erweisen. Zudem wird die Wartbarkeit des Constraint-Wissens reduziert, wenn für einzelne Constraints die Auswertemechanismen separat implementiert werden.[18]

Die Verarbeitung der beschriebenen Tupel-Constraints von EngCon für die extensionale Constraint-Repräsentation wurden sehr pragmatisch umgesetzt, allerdings ist EngCon mit dieser Realisierung auf ein zusätzliches, externes Datenbank-System angewiesen. Unabhängig von der Verfügbarkeit eines solchen Systems, und der damit einhergehenden komplexeren Systemarchitektur, stellt sich die Frage nach der Effizienz dieser Lösung. Über einen längeren Zeitraum wurden mit enormem Aufwand Filter- und Lösungsme-

[18] Unbetroffen bleibt hiervon die Wartbarkeit der Wissensbasis an sich, in der weiterhin eine deklarative Beschreibung erhalten bleibt.

chanismen für die kombinatorische Problemlösung von klassischen Constraint-Problemen entwickelt (vgl. Abschnitt 5.2, S. 83). Diese Verfahren werden auch bzw. gerade für extensionale Constraints verwendet.

Der Schwerpunkt dieser Arbeit liegt auf einem flexiblen Mechanismus zum Auflösen von Funktions- und Prädikat-Constraints in EngCon, wobei die Unterscheidung von Funktions- und Prädikat-Constraints historische Gründe hat. Für das Constraint-System von KonWerk wurden Prädikat-Constraints für sehr große bzw. unendliche Relationen eingeführt, die sich aufgrund ihrer Kardinalität durch Tupel-Constraints nicht mehr effizient definieren lassen. Für die Variablen eines Prädikat-Constraints wurde deren Kreuzprodukt vom Lösungsverfahren gebildet, um die Tupel auf Zugehörigkeit zur durch die Prädikat-Funktion beschriebenen Relation testen zu können. Die Bildung des Kreuzprodukts setzt natürlich in der Praxis wiederum endliche Relationen voraus (vgl. Gülden 1993, S. 54). Funktions-Constraints hingegen wurden bereits in KonWerk als flexibler Mechanismus zur Beschreibung von unendlichen Relation, bzw. Relationen über unendliche Wertemengen, durch Intervall-Funktionen genutzt (vgl. Gülden 1993, S. 39). Von Nachteil ist hier, dass globale Konsistenz für diese Constraints, begründet durch die Lösungsverfahren, i. A. nicht hergestellt werden kann (vgl. Gülden 1993, S. 57 f.). In EngCon werden sowohl Funktions- als auch Prädikat-Constraints von derselben Komponente, dem externen (Intervall-)Constraint-Solver, ausgewertet. Auch diese Komponente erreicht i. d. R. lediglich lokale Konsistenz, setzt allerdings Methoden ein, die in manchen Fällen globale Konsistenz herstellen können.

Vorteilhaft an der derzeit in EngCon eingesetzten Lösung ist, dass sich durch die zum Einsatz kommenden Intervallverfahren sinnvolle Näherungslösungen für ein breites Spektrum von Constraints berechnen lassen. Zudem ist die zurzeit verwendete externe Lösung sehr effizient. Die statische Beschränkung auf eine bestimmte externe Lösung kann sich allerdings nachteilig auswirken, wenn unterschiedliche Problemstellungen unterschiedliche Anforderungen an das Lösungsverfahren stellen, z. B. hinsichtlich der Präzision durch den Grad lokaler Konsistenz. Zudem kann sich die Propagation ausschließlich von Intervallen als ineffizient erweisen, wenn entsprechende Wertebereiche von Constraint-Variablen lediglich sehr wenige, einzelne Lösungen aufweisen, die jedoch untereinander einen sehr großen Abstand voneinander aufweisen. Propagationsverfahren für diskrete Wertebereiche können hier effektiver sein.

3.7.3 Zusammenfassung

Neben dieser spezifischen Kritik am Constraint-System von EngCon lässt sich zusammenfassend feststellen, dass Constraints zwar einen höheren Berechnungsaufwand als alternative Verfahren bei der Auswertung besitzen, dafür lässt sich mit Constraints flexibel und effizient der Lösungsraum einschränken, Konsistenz sicherstellen und Konflikte frühzeitig erkennen. Zudem erleichtern sie die Formulierung der Wissensbasis, da sie eine durch den Menschen „natürlich“ empfundene Repräsentation sind. Durch die deklarative Formulierung gewährleisten sie die Übersichtlichkeit der Wissensbasis für den Wissensingenieur und sorgen somit für eine Verringerung der Fehleranfälligkeit und des Aufwands bei der Formulierung des Wissens.

3.8 Anforderungen an einen Constraint-Solver für EngCon

Nach dieser Übersicht über das Constraint-System von EngCon werden nun die spezifischen Anforderungen aufgezeigt, die für das Konfigurierungswerkzeug EngCon bzgl. eines Constraint-Solvers vorliegen:

Anforderungen an die Schnittstelle Wie bei modernen, wissensbasierten Systemen üblich, sollte eine strikte Trennung von Kontrolle und Constraint-System eingehalten werden, um die Mächtigkeit der Constraints nicht durch eingreifende Steuerung, z. B. bzgl. der Reihenfolge oder der Anzahl der zu verarbeitenden Constraints, zu gefährden (*Black-Box*-Prinzip). Die Steuerung des Constraint-Systems darf sich nur auf die Schnittstelle zu selbigem beziehen (z. B. welcher Solver eingesetzt werden soll), nicht jedoch inhaltlich auf die Propagationsmechanismen der Constraint-Solver (vgl. Günter 1992, S. 77).

Die zu ersetzende bzw. neu zu entwickelnde Komponente für das Constraint-System von EngCon ist ausschließlich der Teil, mit dem Funktions- und Prädikat-Constraints ausgewertet werden. Die vorhandenen übergeordneten Bestandteile, des Pattern-Matching-Mechanismus, die konzeptuellen Constraints, das EngCon eigene, interne Constraint-Netz, sowie die Mechanismen zum Auflösen von Tupel- und Java-Constraints bleiben unangetastet. Es ist notwendig, dass die neue Komponente in der Lage ist, weiterhin im Kontext des bestehenden Constraint-Systems zu arbeiten. Dafür müssen die vorhandenen Schnittstellen von EngCon eingehalten werden.

Die Übergabe der Constraints eines Konfigurierungsproblems in EngCon zur Auswertung an den Constraint-Solver erfolgt derzeit über eine einfache stringbasierte Schnittstelle. Dies ermöglicht eine transparente und intuitive Nutzung durch den Wissensingenieur bei der Modellierung der Wissensbasis. Eine neue Komponente sollte daher vorzugsweise, unabhängig von der Syntax, ebenfalls über einen stringbasierten Zugang verfügen.

Dynamik des Constraint-Netzes Bedingt durch den interaktiven Konfigurierungsverlauf werden in EngCon während einer Konfigurierung durch Zerlegungen und Spezialisierungen sukzessiv neue Komponenten generiert und zur aktuellen Teilkonfiguration hinzugefügt, bis diese eine konsistente und vollständige Lösung darstellt. Das Constraint-System von EngCon unterstützt dementsprechend das inkrementelle Anwachsen des Constraint-Netzes und die Verwaltung neu hinzugekommener Abhängigkeiten (vgl. Abschnitt 3.6, S. 33 und Abschnitt 3.7.2, S. 42). Ebenso sollte es der zum Einsatz kommende Constraint-Solver aus Gründen der Effizienz ermöglichen, dass bei jedem Konfigurierungszyklus nicht das vollständige Constraint-Netz erneut instantiiert und die bisherige Propagation wiederholt werden muss, sondern lediglich die neuen Abhängigkeiten dem bestehenden Constraint-Netz inkrementell hinzugefügt und die Einschränkung der Wertebereiche fortgesetzt werden kann (vgl. Frühwirth und Abdennadher 1997, S. 41 f.; Marriott und Stuckey 1999, S. 352 ff.).

Das Constraint-System sollte Unterstützung für die Zurücknahme von Konfigurierungsentscheidungen bieten. Da dies sehr aufwendig ist, sollten Fehlentscheidungen möglichst von vorne herein ausgeschlossen bzw. frühzeitig erkannt werden. Das System sollte demnach korrekt und vollständig arbeiten und ausschließlich gültige Wertebereiche liefern, die einer konsistenten Konfiguration entsprechen.[19] Auch außerhalb des Constraint-Systems kann es durch die Komplexität des Konfigurierungsproblems und der Größe der Domäne zu Fehlentscheidungen aufgrund von Heuristiken oder Benutzereingaben im Verlauf einer Konfigurierung kommen. Häufig müssen Entscheidungen getroffen werden, ohne dass alle beeinflussenden Informationen bereits vorliegen. Für die Rücknahme von Konfigurierungsentscheidungen muss es möglich sein, den alten Zustand des Constraint-Netzes wiederherzustellen.

Zu unterstützende Domänen Während einer Konfigurierung in EngCon müssen sowohl Elemente aus diskreten Werte*mengen* ausgewählt, als auch kontinuierliche Werte*bereiche* eingeschränkt werden können. Ein Constraint-System, welches ebensolche Konfigurierungsschritte unterstützt, muss infolgedessen Propagations- und Lösungsmechanismen sowohl für diskrete als auch kontinuierliche (Intervall-)Domänen aufweisen.

In technischen Domänen ist es häufig erforderlich, Constraints mit hoher Genauigkeit zu berechnen. Der in EngCon derzeit eingesetzte, externe Constraint-Solver bewerkstelligt dies, indem die Wertebereiche der Constraint-Variablen als reellwertige Intervalle repräsentiert werden. Die Wertebereiche weisen demnach kontinuierliche Intervalldomänen, d. h. eine unendlich große, nicht abzählbare Anzahl von Elementen auf. Diese Repräsentation ist sehr flexibel, so dass sich alle benötigten Berechnungen auf Funktions-Constraints durchführen lassen. Diese Lösung weist jedoch einige Nachteile auf:

- Der Aufwand zur Auswertung von Intervall-Constraints ist, besonders wenn die eingesetzten Constraint-Solver mit hoher Genauigkeit arbeiten, sehr hoch und oftmals unnötig. Häufig sind viele Abhängigkeiten eher trivial und die Wertebereiche der Constraint-Variablen weisen diskrete Domänen mit einer endlichen, abzählbaren Anzahl Elemente auf.
- Die Verarbeitung derartiger Constraints bzw. Wertebereiche durch Constraint-Solver auf Intervallbasis, kann zu Ungenauigkeiten führen, wenn große „Abstände" zwischen einzelnen, diskreten Elementen eines Wertebereichs zu approximierten Ergebnissen des Intervall-Solvers führen.
- Wenn stattdessen die diskreten Elemente der Wertebereiche zur Verarbeitung durch einen Intervall-Solver jeweils als separates Intervall repräsentiert werden, besteht die Gefahr der Ineffizienz.
- Intervall-Constraint-Solver sind i. A. auf numerische Wertebereiche beschränkt. Constraint-Solver für diskrete Wertebereiche sind dagegen in der Lage, auch symbolische Domänen zu verarbeiten.

[19] Aus Komplexitätsgründen kann dies nicht immer eingehalten werden.

Sofern diskrete Wertebereiche einen bestimmten Umfang nicht überschreiten, lassen sich Abhängigkeiten zwischen diesen effektiv durch Tupel-Constraints extensional repräsentieren und verarbeiten. Bei größeren Wertebereichen ist diese Repräsentationsform allerdings ineffektiv und wartungsintensiv. Um die Abhängigkeiten zwischen Komponenten mit umfangreichen Wertebereichen adäquat repräsentieren und verarbeiten zu können, sind eine intensionale Repräsentation und entsprechende Constraint-Solver sowohl für diskrete als auch kontinuierliche Wertebereiche erforderlich.

Eigenschaften von Lösungsverfahren Anforderungen an die Eigenschaften der Lösungsalgorithmen lassen sich nur allgemein formulieren, da ein Problem auf unterschiedliche Art und Weise als Constraint-Problem formalisiert werden kann. Je nachdem, wie dies innerhalb der Wissensbasis von EngCon geschieht, und welche Eigenschaften das resultierende Constraint-Netz aufweist, können sich unterschiedliche Lösungsverfahren als effektiv erweisen (vgl. Abschnitt 4.3, S. 54). Um größtmögliche Flexibilität bzgl. der zu verarbeitenden Constraints bieten zu können, sollte ein Constraint-System n-äre und nichtlineare Constraints sowie zyklische Strukturen des Constraint-Netzes unterstützen. Außerdem sollte das System bzgl. diskreter Wertebereiche nicht auf numerische Werte beschränkt sein, sondern ebenfalls die Möglichkeit bieten, symbolische Domänen einzuschränken.[20] Bezogen auf kontinuierliche Intervalldomänen muss unterschieden werden zwischen ununterbrochenen und unterbrochenen Intervallen (vgl. Abschnitt 5.3.1.4, S. 149). Die Möglichkeit zur Verarbeitung von Constraints bzw. Wertebereichen ist für beide Fälle wünschenswert, wobei sich Lösungen für ununterbrochene Intervalle auf Kosten der Vollständigkeit der Lösungen effizienter realisieren lassen, als für unterbrochene Intervalle.

Präzision versus Effizienz Dem auf der vorherigen Seite bereits genannten Erfordernis an die Präzision von Lösungen steht oftmals der Wunsch nach Effizienz bei den zum Einsatz kommenden Lösungsverfahren und die Komplexität der Problemstellung entgegen. Das Constraint-System sollte demnach einen Kompromiss zwischen Effizienz und Qualität bzgl. der zu berechnenden Lösungen erlauben. Je nach Anwendungsdomäne, den vorhandenen Constraints, der bevorzugten Präzision und der benötigten Effizienz kann es sinnvoll sein, unterschiedliche Verfahren zur Auswertung der Constraints einzusetzen. Ein solches System sollte mehrere Constraint-Solver zur Auswahl bieten, von denen je nach Anforderung einer oder eine Kombination von Solvern ausgewählt werden kann. Bei Einschränkungen bzgl. der Präzision von Lösungsverfahren ist zu beachten, dass keine möglichen Lösungen verloren gehen dürfen, da sonst die Vollständigkeit des Verfahrens nicht gewährleistet ist.[21]

Performance Grundsätzlich ist die Ausführungsgeschwindigkeit der Propagation abhängig vom Umfang des Constraint-Netzes. Je nach Wissensbasis und der Art und Weise, wie die Abhängigkeiten bzgl. der Konfigurierung modelliert wurden, sollten

[20] Sofern sich diese durch algebraische Ausdrücke intensional formulieren lassen.

[21] Ausgenommen der spezielle Fall, in dem durch Benutzerwunsch festgelegt ist, dass z. B. aus Effizienzgründen ausschließlich die erste gefundene Lösung für ein Problem zur Weiterverarbeitung übernommen werden soll.

Constraint-Solver für EngCon in der Lage sein, die Relationen in Constraint-Netzen mit bis zu mehreren hundert Variablen in einer akzeptablen Zeit zu berechnen. Akzeptabel ist für den Benutzer, der interaktiv eine Konfigurierung am Rechner vornimmt, lediglich eine Zeitspanne von wenigen Sekunden.

Je komplexer die modellierten Constraints, umso aufwendiger gestaltet sich deren Auswertung und umso mehr verlängert sich die Dauer der Propagation. Die Ausführungsgeschwindigkeit ist zudem abhängig von der eingesetzten Rechnerplattform bzw. den dort vorkommenden unterschiedlichen Hardware-Ausstattungen. Für den Betrieb von EngCon kann der Einsatz relativ aktueller „Standard"-Hardware vorausgesetzt werden.

Voraussetzungen der Hard- und Software-Umgebung Der EngCon-Konfigurator ist vorrangig für die „Microsoft-Windows-NT"-Plattform auf aktueller x86-Hardware implementiert worden. Eine neue Constraint-Komponente muss daher ebenfalls auf dieser Plattform lauffähig sein.[22]

Vorzugsweise, um den Integrationsaufwand möglichst gering zu halten, sollte die neue Komponente wie auch EngCon in der objektorientierten Programmiersprache Java implementiert sein oder über eine entsprechende Schnittstelle verfügen. Eine Implementierung in Java würde zudem eine weitestgehende Plattformunabhängigkeit garantieren, da in Java geschriebene Programme grundsätzlich plattformübergreifend überall dort ausgeführt werden können, wo eine Java-Laufzeitumgebung verfügbar ist. Dies sind neben Microsoft Windows im Wesentlichen diverse Unix-Derivate (Linux, Solaris, AIX, HP-UX, OpenVMS, True64, Mac OS X, FreeBSD, OpenBSD, NetBSD, etc.), die auf unterschiedlichen Hardware-Architekturen lauffähig sind.

Um auf zukünftige Anforderungen flexibel reagieren und entsprechende Erweiterungen vornehmen zu können, ist es notwendig, über den Quellcode der Constraint-Komponente zu verfügen. Dieser sollte in einer erweiterbaren, modularen Form vorliegen.

Nach dieser Einführung in die Anwendungsdomäne, der wissensbasierten Konfigurierung mit EngCon, und den dargelegten Anforderungen an einen Constraint-Solver zur Unterstützung des Konfigurierungsprozesses, werden in dem folgenden Abschnitt die grundlegenden Formalismen und Konzepte zur Constraint-Verarbeitung dargelegt, sowie bestehende Constraint-Systeme hinsichtlich ihrer Eignung für eine Integration in das Konfigurierungswerkzeug EngCon untersucht.

[22]Die Windows-NT-Produktfamilie von Microsoft für x86-Systeme besteht derzeit aus Windows NT 4.0, Windows 2000 und Windows XP.

Teil II

Grundlagen zur Constraint-Verarbeitung

Kapitel 4

Constraints – Konzepte und verfügbare Systeme

fct . < . < . = (**nat**, **nat**, **nat**) bool *Mixfix*

MANFRED BROY, INFORMATIK, TEIL 1, 1992

Dieser Teil der Arbeit enthält eine Einführung in den Constraint-Formalismus und Übersichten sowohl zu den Eigenschaften von Constraints als auch zu den bestehenden Konzepten zur Constraint-Verarbeitung. Außerdem beinhaltet dieser Abschnitt eine ausführliche Evaluierung und Bewertung existierender Constraint-Systeme hinsichtlich ihrer Eignung für eine Integration in das Konfigurierungswerkzeug ENGCON.

4.1 Einführung

Die Verwendung von Constraints ist eine vielfach eingesetzte Methode zur Repräsentation und Auswertung von Abhängigkeiten. Durch Constraints werden Relationen zwischen (Constraint-)Variablen definiert. Neben der Definition solcher „Beschränkungen" werden Constraints eingesetzt, um die Werte von Variablen dynamisch den Anforderungen der Constraints entsprechend anzupassen (vgl. Syska und Cunis 1991, S. 78). Constraints sind in diesem Sinne Randbedingungen, welche die Konsistenz der Variablenwerte in einem System sicherstellen. Sie können je nach Betrachtungsweise z. B. als Kontrollstruktur oder Suchproblem angesehen werden.

Ein Constraint besteht aus einer endlichen Menge von Variablen, die über eine Relation zueinander in Beziehung gesetzt werden. Jede Variable besitzt einen eigenen Wertebereich (*Domäne*). Der Lösungsraum ist das kartesische Produkt dieser Wertebereiche, und die

Lösungsmenge selber eine Teilmenge dieses Raumes. Formal wird ein Constraint wie folgt definiert (vgl. Güsgen 2000, S. 268):[1]

Definition 4.1.1 (Constraint)
Ein k-stelliges Constraint C ist ein Paar bestehend aus einer Menge von Variablen $v_1, \ldots, v_k$ *und einer entscheidbaren Relation R, wobei die* v_i*,* $i = 1, \ldots, k$*, Werte aus gegebenen Wertebereichen* D_i *annehmen können und R eine Teilmenge von* $D_1 \times \cdots \times D_k$ *ist.*

Constraints werden anhand der Anzahl der jeweils involvierten Variablen unterschieden. Constraints, die jeweils nur eine einzige Variable betreffen, d. h. im Regelfall einfache Wertezuweisungen sind, heißen unäre Constraints. Sind zwei Variablen zueinander in Relation gesetzt, so ist dies ein binäres Constraint, bei drei Variablen ein ternäres etc.

Sind sämtliche Abhängigkeiten zwischen einer Menge von Variablen jeweils in einem einzigen Constraint zusammengefasst, spricht man von totalen Constraints (vgl. Faltings 1994, S. 365):

Definition 4.1.2 (totales Constraint)
Wenn zwischen einer Menge von Variablen $v_1, \ldots, v_k$ *jeweils nur ein einziges Constraint C existiert, welches sämtliche Abhängigkeiten zwischen* $v_1, \ldots, v_k$ *beschreibt, so ist C ein totales Constraint (engl. total constraint).*

Existieren mehrere Constraints, die über ihre Variablen zueinander in Beziehung stehen, d. h. Variablen aus einem Constraint sind gleichzeitig über eine Relation mit Variablen in einem anderen Constraint in Beziehung gesetzt, spricht man von einem *Constraint-Netz*, erstmals formalisiert von Montanari (1974). Ein Constraint-Netz ist also eine Menge von Randbedingungen für eine Menge von Variablen (vgl. Güsgen 2000, S. 268):

Definition 4.1.3 (Constraint-Netz)
Ein Constraint-Netz auf den Variablen $v_1, \ldots, v_n$ *ist eine Menge von Constraints* $C_1, \ldots, C_m$*, so dass die Variablen jedes Constraints eine Teilmenge von* $v_1, \ldots, v_n$ *sind.*

Neben den in Abschnitt 3.6.2 auf Seite 36 ff. bereits eingeführten EngCon-spezifischen Constraint-Arten Funktions-/Prädikat- und Tupel-Constraints, sowie den flexiblen Java-Constraints, gibt es viele weitere Arten von Constraints, die jeweils für bestimmte Domänen besonders geeignet sind. So werden z. B. spezielle *räumliche Constraints* zur Unterstützung des räumlichen Konfigurierens verwendet, *temporale Constraints* zur Beschreibung von Abhängigkeiten zwischen Zeitpunkten bzw. -intervallen bei Zeitplanungsproblemen (*Scheduling*), *Sequenz-Constraints* zur Repräsentation von DNA-Strängen in der Genforschung, *Blocks-World-Constraints* zur Planungsunterstützung, *boolesche Constraints*, in denen die Constraint-Variablen nur zwei Zustände annehmen können und über

[1] Die Notation der in diesem und den nachfolgenden Kapiteln aufgeführten Definitionen und Algorithmen wurde z. T. zugunsten einer einheitlichen Darstellung an die in dieser Arbeit verwendete Notation angepasst.

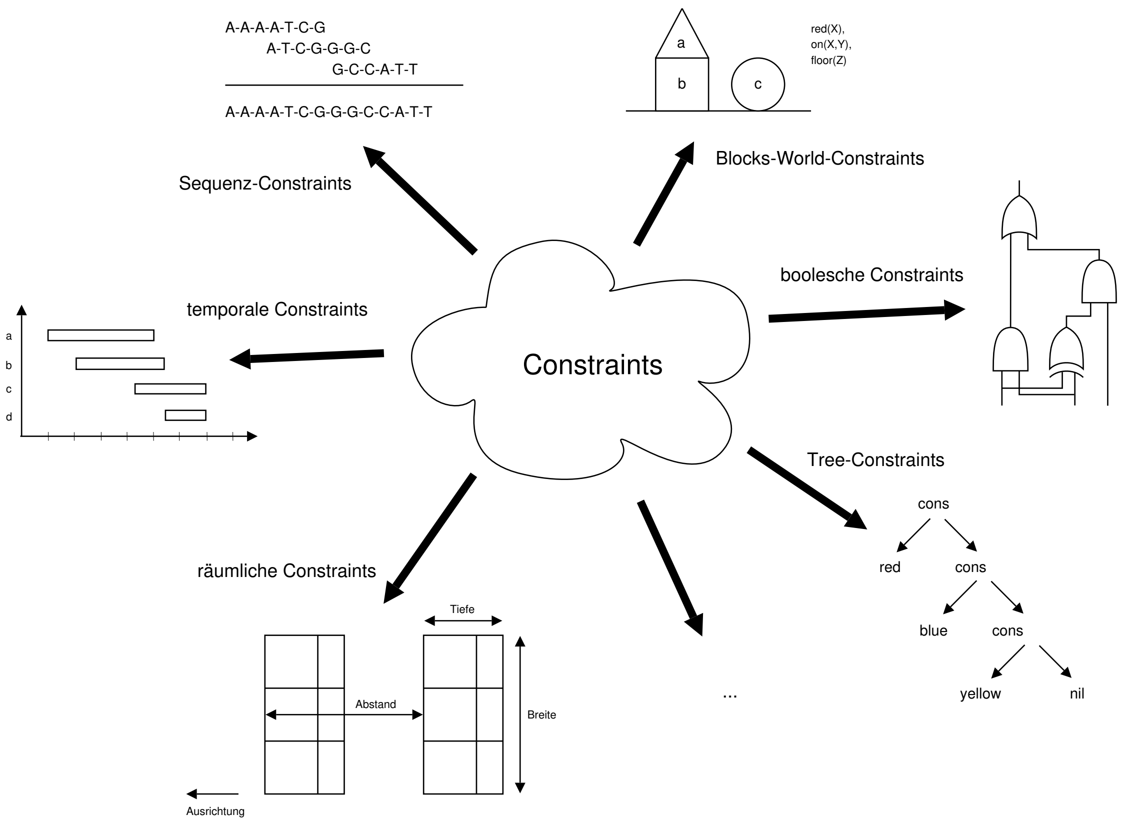


Abbildung 4.1: Beispiele für unterschiedliche Constraint-Arten

boolesche Operatoren verknüpft werden oder Constraints über bestimmte Datenstrukturen, z. B. über Bäume, sog. *Tree-Constraints* (vgl. Günter 1995b, S. 129 ff.; Marriott und Stuckey 1999, S. 22 ff.; Tsang 1993, S. 17 ff.). Alle aufgezählten Domänen bzw. Datenstrukturen weisen unterschiedliche Eigenschaften auf (Domänenelemente, Relationen), die von den jeweils eingesetzten speziellen Constraint-Solvern zum Auflösen der Constraints berücksichtigt werden.

Die Nutzung von Constraints wurde initiiert von *David L. Waltz*, der Constraints zur Analyse von Polyeder-Zeichnungen nutzte (vgl. Waltz 1975). Neben Systemen zur Konfigurierung setzen seither zahlreiche Anwendungen Constraints ein, z. B. aus den Bereichen Analyse, Diagnose, Planung und zeitlichem Schließen (vgl. Freuder 1997; Güsgen 2000). Einen besonderen Stellenwert nimmt der Einsatz von Constraints für die logische Programmierung, insbesondere mit der Programmiersprache Prolog, ein. Dieser Zweig hat sich im Laufe der Zeit zu einem beinahe eigenständigen Forschungsgebiet entwickelt (vgl. Güsgen 2000, S. 267). Die Idee der Logikprogrammierung, bzw. deklarativer Programmierung im Allgemeinen, dass durch den Programmierer festgelegt wird *was* gelöst werden soll anstatt *wie* etwas gelöst werden soll, steht dem Gedanken von Constraints sehr nahe (vgl. Barták 1999b, S. 8). So werden beim *Constraint Logic Programming* (CLP) Constraints als eine Erweiterung der logischen Programmierung gesehen:[2] *„Constraint Logic Programming began as a natural merger of two declarative paradigms: constraint solving and logic programming“* (Jaffar und Maher 1994, S. 503).

Spezifische Constraints, die im Bereich *Constraint Programming* (CP), d. h. allgemein für die Programmierung von Lösungen für spezielle kombinatorische Problemstellungen mittels Constraint-Technologien, von besonderer Relevanz sind, werden von Systemen zur (logischen) Constraint-Programmierung explizit zur Verfügung gestellt. Die Propagationsregeln für diese Constraints werden i. d. R. separat und besonders effektiv implementiert.[3] Diese Art Constraints werden auch *global constraints* genannt, da sie i. A. mehr als zwei Variablen umfassen und dazu dienen, eine Menge kleinerer Constraints zur effektiveren Propagation zusammenzufassen. Die meisten CP- bzw. CLP-Systeme verfügen über eine Reihe von diesen fest vorgegebenen *global constraints* (vgl. Barták 2001, S. 12; Dechter und Rossi 2003, S. 798).

4.2 Algebraische Constraints

Eine allgemeine Form von Constraints sind *algebraische Constraints*. Sie bezeichnen beliebige algebraische Ausdrücke, über die involvierte Constraint-Variablen intensional, d. h. implizit über eine Gleichung bzw. Ungleichung, zueinander in Beziehung gesetzt werden. Diese intuitive und gleichzeitig mächtige Form von Constraints entspricht damit den Funktions- und Prädikat-Constraints in der EngCon-Terminologie. Im Folgenden bezieht sich der Ausdruck *Constraint* daher auf algebraische Constraints, sofern nicht gesondert hervorgehoben.

Constraints lassen sich unterscheiden in Constraints mit Wertebereichen in endlichen (*finiten*) und unendlichen (*infiniten*) Domänen. Zur Unterstützung des Konfigurierungsprozesses in EngCon mit algebraischen Constraints ist die Verarbeitung sowohl von finiten als auch von infiniten Domänen notwendig.

Die Wertebereiche von Constraints über finite Domänen lassen sich aufzählen, um eine Lösung bzw. alle möglichen Lösungen zu finden, d. h. sie sind *diskret*. Allerdings lässt sich u. U. nicht für jede aufzählbare Domäne eine Lösung finden.[4] Constraints über **finite Domänen** haben dementsprechend Wertebereiche, die **diskret** und **endlich** sind. Diese Bedingung trifft aber nicht ausschließlich auf algebraische Constraints mit Zahlenwerten als Domänenelemente zu. So sind z. B. boolesche Constraints und *Blocks-World*-Constraints ebenfalls Constraints über finite Domänen. Die einzelnen Elemente der Wertebereiche müssen daher nicht unbedingt arithmetische Werte sein. Ebenso können auf-

[2] Der Prozess des *Pattern-Matching* während der Unifikation von Prolog wird durch *Constraint Satisfaction* erweitert bzw. ersetzt und erlaubt dadurch eine effiziente Verarbeitung von algebraischen Constraint-Ausdrücken innerhalb von logischen Programmiersprachen (vgl. Cohen 1990; Pountain 1995).

[3] Beispielsweise `all-different`, `cummulative`, `element` (vgl. Marriott und Stuckey 1999, S. 110 ff.).

[4] Wenn die Wertebereiche der Constraint-Variablen z. B. die natürlichen Zahlen sind, findet der eingesetzte Constraint-Solver möglicherweise niemals eine Lösung. Je nachdem, nach welchem Algorithmus der Constraint-Solver arbeitet, bzw. welche Heuristik für die Reihenfolge der Aufzählung verwandt wird, lassen sich hier ggf. trotzdem Lösungen generieren.

zählbare, symbolische Domänen, wie z. B. $\{rot, grün, blau\}$, über algebraische Ausdrücke in Beziehung zueinander gesetzt werden.[5]

Neben Eigenschaften, deren mögliche Ausprägungen sich durch eine aufzählbare, endliche Menge beschreiben lassen, können die zulässigen Eigenschaftswerte zur Modellierung von Systemen in technischen Domänen häufig nur mittels reellwertiger Intervalle adäquat beschrieben werden. Neben algebraischen Constraints mit finiten Domänen werden in dieser Arbeit daher Constraints mit infiniten Domänen betrachtet. Diese Constraints, auch reellwertige algebraische Constraints genannt, sind in Domänen eingebettet, deren Wertebereiche nicht abzählbar und dementsprechend *kontinuierlich* sind.[6] Solche Wertebereiche werden in Form von Ober- und Untergrenzen von Intervallen definiert, zwischen denen unendlich viele reelle Zahlen als Teil der Lösung liegen können. Diese Intervalle können zudem „Lücken“ aufweisen bzw. unterbrochen sein; d. h. kontinuierliche Wertebereiche können aus mehreren kontinuierlichen Abschnitten bestehen. Die Wertebereiche von Constraints über **infinite Domänen** sind folglich **kontinuierlich** und **unendlich**. Zum einen ermöglicht dies, reellwertige Intervalle als Wertebereich von Constraint-Variablen anzugeben, zum anderen kann auf diesem Wege höchstmögliche Präzision bei der Berechnung mit reellen Zahlen erreicht werden, die sich auf einem Rechner nur mit einer begrenzten Genauigkeit darstellen lassen.

4.3 Eigenschaften von Constraints

Constraints bzw. Constraint-Netze können anhand unterschiedlicher Eigenschaften charakterisiert werden (vgl. Tsang 1993, S. 46 ff.). Diese spezifischen Eigenschaften können zur Effizienzsteigerung durch Lösungsverfahren ausgenutzt werden. Bei der späteren Diskussion der Effektivität einzelner Lösungsverfahren bzw. -kombinationen bzgl. unterschiedlicher Problemstellungen wird auf diese Eigenschaften Bezug genommen.

Üblicherweise werden eine Reihe von allgemeinen Eigenschaften betrachtet. Neben der bereits angesprochenen Unterscheidung in Constraints mit **finiten** und **infiniten Domänen** und **symbolischen** und **numerischen Wertebereichen**, zählt dazu die **Größe eines Problems**. Sie betrifft sowohl die Anzahl der Variablen, den Umfang der Wertebereiche als auch die Anzahl der Constraints. Weiterhin ist häufig die **Stelligkeit der Constraints** von besonderem Interesse. Binäre Constraints setzen maximal zwei Constraint-Variablen zueinander in Relation. Bei allgemeinen Constraints ist diese Anzahl nicht begrenzt. Der Hintergrund ist, dass sich binäre Constraints im Gegensatz zu allgemeinen Constraints effektiver Propagieren lassen, da immer nur die Werte für eine Variable innerhalb eines Constraints bekannt sein müssen, um Werte für die jeweils andere ableiten zu können.

Die **Stelligkeit einer Variable** bezeichnet hingegen den Grad ihres Vorkommens in den Constraints, d. h. von wie vielen Constraints eine Variable beschränkt wird.

[5]Während z. B. Operatoren wie = und ≠ unproblematisch sind, müssen für weitergehende bzw. komplexere Relationen die jeweils benötigten, speziellen Operatoren sowie Operationen implementiert werden, um Anwendung finden zu können.

[6]Kontinuierlich im Sinne eines kontinuierlichen mathematischen Problems.

Für die Art der einzusetzenden Lösungsverfahren ist entscheidend, wie das **gewünschte Ergebnis** aussehen soll. So muss unterschieden werden, ob lediglich ein bestimmter Konsistenzgrad hergestellt werden soll, oder ob eine, eine bestimmte/optimale oder alle Lösungen benötigt werden. Je höher der hergestellte Konsistenzgrad (bei entsprechendem Aufwand), umso leichter fällt die anschließende Suche nach Lösungen, da vorher bereits eine größere Anzahl möglicher Variablenbelegungen als ungültig *herausgefiltert* wurde. Für global konsistente Constraint-Netze kann eine Lösung in linearer Zeit bestimmt werden (vgl. Abschnitt 5.2.3.5, S. 103).

Von besonderem Interesse bzgl. der Effektivität von Lösungsverfahren ist ebenfalls die **Struktur des Constraint-Netzes**: Wenn ein Netz z. B. eine baumartige Struktur aufweist, oder in eine solche konvertiert werden kann, lässt sich das Problem in Polynomialzeit lösen (vgl. Abschnitt 5.2.3.5, S. 105). Wenn dies nicht der Fall ist, bzw. nicht möglich ist, bieten sich einige strukturelle Anhaltspunkte, an der die Effektivität bestimmter Lösungsverfahren festgemacht werden kann:

- **Constraint-Dichte:** Es wird zwischen Constraint-Netzen mit hoher Constraint-Dichte (engl. *high density*) und niedriger Constraint-Dichte (engl. *low density* bzw. *sparseness*) unterschieden. Problemgeneratoren für Constraint-Solver bspw. beschreiben mit der Constraint-Dichte das Verhältnis von vorhandenen zu möglichen Constraints zwischen den Constraint-Variablen.[7] Wenn sämtliche Variablen jeweils durch ein Constraint miteinander in Relation gesetzt werden, spricht man von einem vollständigen Netz, bzw. vollständiger Vernetzung des Constraint-Netzes.

- **Beschränkungsgrad:** Der Grad der Beschränkung durch Constraints (engl. *constrainedness* bzw. *tightness*) wird durch die „Lösungsdichte“, das Verhältnis der tatsächlichen zu möglichen Lösungen, definiert. Die Lösungsdichte eines Constraints bzw. eines Constraint-Netzes ist das Verhältnis der Anzahl der Tupel, die das Constraint bzw. das Constraint-Netz erfüllen, zur Anzahl der möglichen Tupel, d. h. zur Kardinalität des kartesischen Produkts der beteiligten Variablen. Je mehr Lösungen ein Constraint bzw. ein Constraint-Problem aufweist, umso höher ist die Lösungsdichte und umso schwächer die Beschränktheit (engl. *loosely constrainedness*). Entsprechend ist bei starker Beschränkung (engl. *tightly constrainedness*) die Lösungsdichte niedriger.

- **Schwierigkeitsgrad:** Der Schwierigkeitsgrad bzw. die „Härte“ (engl. *hardness*) eines Constraint-Problems ist sowohl abhängig von dem Grad der Beschränkung, bzw. der Lösungsdichte, als auch von der Anzahl der benötigten Lösungen. Ein Problem ist einfach, wenn es eine hohe Lösungsdichte aufweist, und lediglich irgendeine Lösung benötigt wird (engl. *easyness*). Ein derartiges Problem ist umso härter, je höher der Grad der Beschränkung und entsprechend umso niedriger die Lösungsdichte ist, da potentiell mehr Suchaufwand nötig wird, eine Lösung zu finden. Wenn sämtliche Lösungen eines Constraint-Problems benötigt werden, ist es umso härter,

[7] Es wird in diesem Fall davon ausgegangen, dass zwischen einer Menge von Variablen jeweils nur ein einziges (totales) Constraint besteht, welches sämtliche Abhängigkeiten zwischen genau diesen Variablen beschreibt (vgl. Definition 4.1.2, S. 51).

wenn viele Lösungen vorliegen, d. h. die Beschränkung entsprechend niedrig ist, da in diesem Fall ein großer Suchraum durchsucht werden muss. Bei hoher Beschränkung kann ein solches Constraint-Problem durch geeignete Lösungsverfahren ggf. relativ einfach zu lösen sein, da durch entsprechende Maßnahmen zur Problemreduktion weniger Suchaufwand betrieben werden muss.

Manche Lösungsmechanismen sind dementsprechend besser geeignet für Constraint-Probleme mit höherem Beschränkungsgrad, andere eignen sich besser für Probleme mit niedrigerer Beschränkung. Problemreduktion ist umso effektiver, je einschränkender die Constraints sind. Dies ist, neben der Anzahl der benötigten Lösungen, ein Grund dafür, weshalb stark beschränkte Constraint-Probleme nicht unbedingt härter zu lösen sein müssen als weniger stark beschränkte.

Bei schwach beschränkten Problemen sind im Verhältnis viele Blätter des Suchbaums Lösungen für das Constraint-Problem. Auch einfache Suchverfahren stoßen daher i. A. bereits frühzeitig auf eine Lösung. Verfahren zur Problemreduktion bzw. Konsistenzherstellung sind in diesem Fall unnötiger Overhead. Sollen sämtliche Lösungen ermittelt werden, wird ein schwach beschränktes Problem schnell sehr „hart“, denn aufgrund der vielen vorhandenen Lösungen muss durch den Mechanismus zur Lösungssuche ein in diesem Fall größerer Suchraum durchlaufen werden.

Je stärker beschränkt ein Problem ist, umso aufwendiger ist es für naive Suchalgorithmen, Lösungen zu finden. Ein stark beschränktes Problem ist daher härter, verglichen mit schwach beschränkten Problemen, wenn es darum geht, nur eine Lösung zu finden. Um alle Lösungen zu ermitteln, kann bei starker Beschränkung sinnvoll mit Mitteln zur Problemreduktion gearbeitet werden, so dass im Endeffekt ein geringerer Suchraum vom Lösungsalgorithmus durchlaufen werden muss (vgl. Tsang 1993, S. 49).

Analog zum Grad der Beschränkung und entsprechend der Anzahl der vorhandenen Lösungen werden Constraint-Probleme in unter-, über- und wohlbestimmte Probleme unterschieden (vgl. Cheeseman et al. 1991, S. 331). Ein Constraint-Problem P heißt

- **unterbestimmt**, wenn P mehrere Lösungen aufweist (engl. *under-constrained*),
- **überbestimmt**, wenn P keine Lösung aufweist (engl. *over-constrained*),
- **wohlbestimmt**, wenn P genau eine Lösung aufweist (engl. *well-constrained*).

Konfigurierungsprobleme sind in der Ausgangssituation häufig unterbestimmt (vgl. Fleischanderl et al. 1998, S. 66). In Konfiguratoren, die Benutzerinteraktionen zulassen, bzw. durch Konfigurierungskonflikte können allerdings überbestimmte Situationen entstehen. Für die Behandlung überbestimmter Constraint-Netze gibt es spezielle Konzepte, für die ein Überblick in Abschnitt 4.4.3 auf Seite 59 ff. gegeben wird.

Der Übergang zwischen unter- und überbestimmten Constraint-Problemen wird auch (engl.) *phase transition region* genannt. Dies ist der Bereich zwischen der Menge von Problemen, die fast alle viele Lösungen aufweisen und einfach zu lösen sind, und Problemen, die fast ausnahmslos keine Lösung aufweisen und deren Unlösbarkeit relativ einfach nachzuweisen ist. In der **Phasenübergangsregion** ist dementsprechend ein abrupter

Wechsel der Lösungswahrscheinlichkeit für Probleme von nahezu 0 nach annähernd 1 zu beobachten (vgl. Cheeseman et al. 1991, S. 335). Probleme in diesem Übergangsbereich sind durchschnittlich relativ hart, d. h. mit höherem Schwierigkeitsgrad behaftet, und nur mit erhöhtem Aufwand zu lösen. Einzelne, besonders harte Probleme (engl. *exceptionally hard problems*) treten allerdings eher in Regionen auf, wo sich ansonsten eher leicht zu lösende Probleme befinden. Dies ist allerdings wiederum abhängig von den eingesetzten Lösungsverfahren, so dass formuliert werden kann, dass Probleme, die für bestimmte Lösungsalgorithmen besonders hart zu lösen sind, für andere Algorithmen sehr leicht zu lösen sein können (vgl. Grant und Smith 1995, S. 2 f.; Grant und Smith 1996, S. 175).

Zur Beantwortung der Frage, welche Lösungsverfahren für welche Problemstellungen sinnvoll eingesetzt werden können, werden neben theoretischen Überlegungen und *worst-case* Analysen überwiegend empirische Untersuchungen herangezogen, da *worst-case* Analysen ihrer Natur nach pessimistisch sind und häufig nicht die Praxistauglichkeit eines Algorithmus reflektieren. Üblicherweise werden zur empirischen Evaluierung von Lösungsalgorithmen relativ harte Probleme aus der Phasenübergangsregion mit Hilfe von Problemgeneratoren erzeugt (vgl. Dechter 1999, S. 196 f.; Gülden 1993, S. 33).[8]

Um eine Vielzahl unterschiedlicher Constraint-Probleme effizient Verarbeiten zu können, sind daher ebenso unterschiedliche Constraint-Solver notwendig. In Bezug auf die strukturbasierte Konfigurierung und das Konfigurierungswerkzeug EngCon ist daher eine flexible Lösung anzustreben, die in Abhängigkeit von der Problemstellung den Einsatz von jeweils geeigneten Lösungsalgorithmen ermöglicht.

4.4 Ansätze zur Constraint-Verarbeitung

Die Möglichkeiten den Constraint-Formalismus, d. h. für eine Menge von Variablen konsistente Belegungen bzgl. einer Menge von Constraints zu finden, für unterschiedliche Anwendungen nutzbar zu machen, sind äußerst vielfältig. Dieser Bereich ist bis heute aktiver Forschungsgegenstand. Im Wesentlichen wurden bisher eine Reihe von unterschiedlichen Rahmenkonzepten entwickelt, mit deren Hilfe Constraint-Mechanismen in verschiedenen Anwendungsszenarien eingesetzt werden können. Im Folgenden werden die geläufigsten Ansätze vorgestellt.

4.4.1 Allgemeine Konzepte

Der klassische Ansatz, Constraint-Probleme zu formalisieren, ist die Beschreibung als **Constraint Satisfaction Problem (CSP)**,[9] welches im Wesentlichen auf die Arbeiten von Montanari (1974), Waltz (1975) und Mackworth (1977a) zurückgeht. Im klassischen „Constraint-Erfüllungsproblem“ verfügen Constraint-Variablen über finite Wertebereiche (engl. *finite domains*, FD). Entsprechende Constraints werden daher auch *FD-Constraints* genannt. Ziel ist es, Wertekombinationen für die Constraint-Variablen zu finden, welche sämtliche durch die Constraints formulierten Restriktionen erfüllen.

[8] Problemgeneratoren setzen allerdings i. A. eine extensionale Repräsentation der Constraints und eine entsprechende Verarbeitung durch Lösungsalgorithmen voraus.

[9] *satisfaction* (engl.): Erfüllung, Zufriedenstellung

Beziehen sich die Domänen der Constraint-Variablen auf reellwertige Werte*bereiche* bzw. *-intervalle* mit unterer und oberer Schranke, spricht man von einem **Interval Constraint Satisfaction Problem (ICSP)**, wie es von Hyvönen (1992) beschrieben wird. Constraint-Variablen in ICSPs verfügen über infinite Wertebereiche. Entsprechende Constraints werden *Intervall-Constraints* genannt. Nicht zu verwechseln sind ICSPs mit **Temporal Constraint Satisfaction Problems (TCSP)**, mit deren Hilfe sich Zeitplanungsprobleme als CSP formulieren und verarbeiten lassen (*Scheduling*). In TCSPs werden die kontinuierlichen Wertebereiche von Constraint-Variablen ebenfalls durch Intervalle repräsentiert, in diesem Fall allerdings durch *Zeitintervalle*. Die Definition von TCSPs geht zurück auf die Arbeit von Allen (1983), in der eine qualitative Intervallalgebra mit entsprechenden Relationen für die Anwendung im Bereich des zeitlichen Schließens entwickelt wird (engl. *temporal reasoning*).[10]

4.4.2 Constraint-Verfahren zur Konfigurierung

Seitdem die Nutzung von Constraints im Rahmen von Konfigurierungsaufgaben mehr und mehr an Bedeutung gewonnen hat, wurden spezielle Vorgehensweisen entwickelt, Konfigurierungsaufgaben als CSP abzubilden (vgl. Mittal und Frayman 1989). Allgemein werden Constraints innerhalb einer Konfiguration dazu genutzt, die Art und Weise, wie unterschiedliche Komponenten miteinander kombiniert werden können, einzuschränken. Vor dem Hintergrund sich verändernder Teilkonfigurationen ist für den Bereich der Konfigurierung der dynamische Aspekt von besonderem Interesse. Da das klassische CSP ein statisches Problem ist, wurden weitergehende Ansätze entwickelt und in constraint-basierte Produkte implementiert (vgl. Abschnitt 2.3, S. 17 und Anhang A, S. 260). Ein solcher Ansatz ist das **Dynamic Constraint Satisfaction Problem (DCSP)** von Mittal und Falkenhainer (1990). Das DCSP wurde gegenüber dem herkömmlichen CSP in dem Sinne erweitert, dass unterschiedliche Lösungen für ein und dasselbe Problem weder dieselben Variablen noch dieselben (erfüllten) Constraints beinhalten müssen. Dies wird erreicht, indem Variablen und Constraints den Status „aktiviert“ bzw. „deaktiviert“ annehmen können. Die Bedingungen bzgl. des Aktivierungsstatus von Variablen, d. h. welche der initial nicht aktiven Variablen inkrementell zu welchem Zeitpunkt aktiviert werden, sind durch sog. *activity constraints* formuliert. Ein weiterer Constraint-Typ (*compatibility constraints*) verkörpert herkömmliche Constraints, die ausschließlich dann aktiviert werden, wenn alle darin vorkommenden Variablen aktiviert sind. In einem DCSP nach Mittal und Falkenhainer (1990) werden somit Bedingungen bzgl. der Existenz von Variablen und Constraints selbst wiederum durch (Meta-)Constraints formuliert.[11]

[10] Neben dem Ansatz der qualitativen Constraints zwischen Zeit*intervallen* von Allen (1983) gibt es alternativ die Möglichkeit, metrische Constraints innerhalb von TCSPs zwischen Zeit*punkten* zu betrachten (vgl. Dechter et al. 1991), sowie Kombinationen beider Ansätze (vgl. Gennari 1998; Schwalb und Vila 1998).

[11] Allerdings ist es mit DCSPs nach Mittal und Falkenhainer (1990) nicht möglich, ein Problem in nichtmonotoner Art und Weise zu beschreiben, in der Form, dass Variablen aus dem Constraint-Netz entfernt werden. Stattdessen lösen Aktivitäts-Constraints wie herkömmliche Constraints eine Inkonsistenz aus, wenn eine Variable aktiviert ist, die laut Constraint deaktiviert sein müsste (vgl. Stumptner 1997, S. 117). Die Auflösung derartiger Inkonsistenzen wiederum kann durch ein übergeordnetes Lösungsverfahren ge-

Diese Art dynamischer Constraint-Erfüllung ist nicht zu verwechseln mit dem DCSP, wie es von Dechter und Dechter (1988) beschrieben wird. Laut Dechter und Dechter (1988) ist ein **Dynamic Constraint Satisfaction Problem** P eine Sequenz $P_0 \dots P_\alpha$ von statischen CSPs, wobei jedes nachfolgende Problem aus der Veränderung zum Vorgänger resultiert. Veränderungen sind demnach das Hinzufügen von Constraints und Variablen (engl. *restriction*) bzw. das Entfernen von Constraints (engl. *relaxation*). Dechter und Dechter (1988) nutzen den derart definierten DCSP-Formalismus zum Vergleich von Lösungen, um die Auswirkungen von Änderungen an CSPs aufzuzeigen.[12] Zur Unterscheidung beider Ansätze erfolgte in neueren Veröffentlichungen (vgl. Sabin und Freuder 1998, 1999) eine Umbenennung des DCSP von Mittal und Falkenhainer (1990) in **Conditional Constraint Satisfaction Problem (CondCSP)**.

Eine weitere Variante des klassischen CSP, welches ebenfalls den Bereich der Konfigurierung adressiert, allerdings ohne Nutzung von Aktivitäts-Constraints, ist das **Composite Constraint Satisfaction Problem (CompCSP)**. Das CompCSP erlaubt die Abbildung von Aggregatstrukturen und eine hierarchische Organisation der Komponententypen, indem das klassische CSP-Paradigma in der Hinsicht erweitert wird, dass Werte für eine (Meta-)Variable vollständige Teilprobleme repräsentieren können. Neben der hierdurch möglichen Strukturierung ist durch die Variation unterschiedlicher Teilprobleme ebenfalls ein dynamischer Effekt gegeben. Zudem lassen sich, da keine speziellen Constraints o. ä. verwendet werden, herkömmliche Lösungsalgorithmen für CSPs auf einfache Art und Weise zur Auflösung von CompCSPs übertragen (vgl. Sabin und Freuder 1996a, b).

Eine Weiterentwicklung des DCSP bzw. CondCSP ist das **Generative Constraint Satisfaction Problem (GCSP)** von Stumptner und Haselböck (1993). Das GCSP bietet neben dem dynamischen Aspekt durch Aktivitäts-Constraints im Gegensatz zum CondCSP und CompCSP die Möglichkeit, Constraints auf einem Metalevel als Relationen zwischen Komponenten*typen* anstatt zwischen bestimmten Komponenten zu spezifizieren, wodurch die gleiche Behandlung mehrfach vorkommender Komponenten ermöglicht bzw. vereinfacht wird. Diese sog. *generischen* Constraints enthalten Metavariablen als Platzhalter für Komponenten-Variablen (vgl. Stumptner et al. 1998, S. 314). Generische Constraints sind vergleichbar mit den konzeptuellen Constraints aus dem dreistufigen Constraint-Modell von EngCon, welches ebenfalls ein dynamisch anwachsendes Constraint-Netz unterstützt. Anstatt Aktivitäts-Constraints verfügt EngCon über einen flexiblen Pattern-Matching-Mechanismus zur gesteuerten Instantiierung von Constraint-Relationen (vgl. Abschnitt 3.6, S. 33 und Abschnitt 3.7.2, S. 42).

4.4.3 Überbestimmte Constraint-Probleme

Auch wenn die Problemstellungen im Bereich der Konfigurierung häufig unterbestimmt sind, können sich insbesondere bei inkrementellen Konfiguratoren, die während des Konfi-

schehen (z. B. Backtracking, vgl. Abschnitt 5.2.4.2, S. 114) oder bereits bei der Modellierung des Problems durch Einfügen entsprechender Constraints verhindert werden (vgl. Sabin und Freuder 1999, S. 92).

[12] Dies geschieht zur Unterstützung von *belief* bzw. *reason maintenance* innerhalb von *Truth Maintenance Systems* (TMS), in denen Constraint-Verfahren zur Verwaltung von Abhängigkeiten eingesetzt werden (vgl. Russel und Norvig 2002, S. 157 u. S. 360 f.).

gurierungsprozesses Benutzerinteraktionen zulassen, inkonsistente Zustände ergeben, weil der Benutzer während der Konfigurierung sich widersprechende oder zu starke Anforderungen an die Konfiguration stellt. Eine solche Situation ist überspezifiziert bzgl. der Abhängigkeiten innerhalb der Konfigurierung, die je nach eingesetztem Konfigurator z. T. oder gar vollständig durch Constraints repräsentiert sein können. Im Bereich des Constraint Satisfaction spricht man von überbestimmten Constraint-Systemen bzw. **Over-Constrained Systems (OCS)**. Abhilfe wird durch Verfahren geschaffen, die spezielle Konzepte zur Behandlung derartiger Konflikte anbieten. Da für ein OCS keine Lösungen im herkömmlichen Sinn existieren, wird stattdessen versucht, suboptimale Lösungen durch Teilerfüllung der Beschränkungen zu finden.

Freuder (1989) bzw. Freuder und Wallace (1992) entwickeln hierfür ein allgemeines Rahmenkonzept zur partiellen Constraint-Erfüllung mit dem Namen **Partial Constraint Satisfaction Problem** (PCSP). Freuder (1989, S. 280) führt aus, dass vier Möglichkeiten auf der Ebene eines Constraint-Systems existieren, Inkonsistenzen in einem überbestimmten Constraint-Problem zu entschärfen:

- Erweiterung des Wertebereichs einer Variable um geeignete Werte.
- Erweiterung der Menge R der Relationstupel eines Constraints, die angibt welche Wertekombinationen der Variablen kompatibel sind.
- Eliminierung von Variablen und der zugehörigen Constraints.
- Eliminierung von Constraints.

Unabhängig von dem eingesetzten Verfahren zur (Teil-)Lösung eines OCS wird eine Bewertungsfunktion benötigt, um die für die jeweilige Anwendung möglichst bestgeeignete Lösung auswählen zu können. Diese Bewertungsfunktion, auch Metrik genannt, dient dazu, eine Ordnung auf den unterschiedlichen, suboptimalen Lösungen zu definieren.[13]

Die Eliminierung von Constraints ist auch unter der Bezeichnung **Constraint-Relaxierung** bekannt (vgl. Güsgen 2000, S. 268 u. S. 281).[14] Möglichkeiten zur Constraint-Relaxierung in das Constraint-System zu implementieren, wurden bereits für die Vorgänger von EngCon, PlaKon bzw. KonWerk diskutiert (vgl. Günter 1992, S. 102 f.; Günter 1995b, S. 114; Syska und Cunis 1991, S. 88 f.).

Von Freuder und Wallace (1992) wird im Wesentlichen eine bestimmte Ausprägung des PCSP behandelt, genannt das **Maximal Constraint Satisfaction Problem (MaxCSP)**. Innerhalb eines MaxCSP kommt eine simple Metrik zum Einsatz, die besagt, dass

[13]Eine Metrik ist eine Abstandsfunktion, die in diesem Fall definiert, wie weit die jeweilige suboptimale Lösung von einer vollständigen Lösung „entfernt“ ist. Ziel ist die Minimierung dieses „Abstands“.

[14]Von engl. *to relax*: entspannen, lockern. Gemeint ist die „Lockerung“ bzw. Verringerung des Beschränkungsgrades eines Constraint-Netzes. Nicht zu verwechseln ist dies mit der Relaxierung, wie sie durch Konsistenzverfahren erreicht wird, indem (ungültige) Werte aus den Wertebereichen der Constraint-Variablen eliminiert werden (vgl. Fußnote 14, S. 91). Grundsätzlich ist ein Constraint-Netz *relaxed*, wenn alle Constraints erfüllt sind (vgl. Freuder 1978, S. 963). Dies lässt sich auf unterschiedliche Weise erreichen, sowohl durch die Eliminierung von Constraints als auch durch Konsistenzverfahren und Constraint-Propagation.

eine Problemlösung umso besser ist, je mehr Constraints durch diese Lösung erfüllt werden. Dies kommt ebenfalls der Relaxierung bestimmter Constraints gleich, die nicht Bestandteil einer auf diesem Wege generierten Lösung sind.[15]

Da innerhalb eines PCSP bestimmte Constraints nicht erfüllt werden können, wird das Rahmenkonzept für eine derartige Constraint-Behandlung z. T. auch **Soft Constraint Satisfaction Problem (SoftCSP)** genannt (vgl. Ruttkay 1994). Ein SoftCSP beinhaltet die Unterscheidung in weiche Constraints (engl. *soft constraints*) und harte Constraints (engl. *hard constraints*). Während weiche Constraints im Konfliktfall nicht unbedingt erfüllt werden müssen, um trotzdem zu einer „konsistenten" Lösung zu gelangen, entsprechen harte Constraints den klassischen Beschränkungen, die in jedem Fall erfüllt werden müssen und nicht zurückgenommen werden können.

Eine in diesem Sinne weiterführende Möglichkeit zur Behandlung von überbestimmten Constraint-Netzen besteht in einer differenzierteren Unterscheidung von Lösungen anhand deren Qualität. Dies kann in Form von Constraint-Hierarchien durch **Hierarchical Constraint Satisfaction Problems (HCSP)** geschehen, einer weiteren Ausprägung des PCSP (vgl. Borning et al. 1987, 1992, 1994). Das HCSP erlaubt ebenfalls eine Relaxierung bestimmter Constraints, in diesem Fall indem eine Gewichtung entsprechend einer endlichen Menge unterschiedlicher Prioritätsstufen vorgenommen wird. Innerhalb dieser Hierarchie sind Constraints nach Präferenz geordnet. Constraints der höchsten Stufe müssen erfüllt werden, während niedriger priorisierte Constraints verletzt werden dürfen. Eine lexikographische Ordnung der Hierarchiestufen bestimmt die Güte unterschiedlicher Lösungen anhand der Gewichtungen der erfüllten Constraints. Die Erfüllung eines höher gewichteten Constraints wird dabei immer höher gewertet als die Erfüllung beliebig vieler geringer gewichteter Constraints auf niedrigeren Hierarchiestufen.[16] Im Gegensatz zum MaxCSP von Freuder und Wallace (1992), deren Metrik sich auf den Problemraum bezieht, d. h. auf die unterschiedlichen Abstände von relaxierten Problemen zum Originalproblem, bezieht sich die Metrik in einem HCSP auf den Lösungsraum, da hier die Güte unterschiedlicher Lösungen bewertet wird (vgl. Borning et al. 1992, S. 261).[17]

Die Arbeiten von *Alan Borning* et al. führten zu einer Reihe unterschiedlicher Lösungsalgorithmen für hierarchische Constraint-Probleme namens „Blue", „DeltaBlue", „DeltaStar", „SkyBlue", „Indigo", „Ultraviolet" und „Cassowary", deren Anwendungsbereich überwiegend in der deklarativen Behandlung von Einschränkungen im Bereich constraint-basierter grafischer Benutzungsoberflächen angesiedelt ist. Die bisher bekannten Lösungsverfahren für HCSPs sind z. T. inkrementell nutzbar. Sie alle weisen jedoch einige gravierende Einschränkungen auf: So können die zum Einsatz kommenden Lösungsalgorithmen entweder zyklische Constraint-Netze gar nicht oder nur eingeschränkt verarbeiten (*local propagation*), oder sie sind zwar in der Lage, auch zyklische Constraint-Netze zu verarbei-

[15] Freuder und Wallace (1992) setzen ein vollständiges, systematisches Suchverfahren namens *Branch & Bound* zur Optimierung, d. h. zur Maximierung der Anzahl der erfüllten Constraints, ein.

[16] Dies ist die klassische Variante eines HCSP. In neueren Veröffentlichungen werden weiterführende Konzepte zum Vergleich erfüllter Constraints auf unterschiedlichen Hierarchiestufen dargelegt.

[17] Güsgen (1989, S. 51 f.) und Gülden (1993, S. 49 ff.) benennen mit Constraint-Hierarchien für ConSat bzw. für das Constraint-System von KonWerk die Bündelung einfacher Constraints zu zusammengesetzten Constraints. Bei der Propagation ermöglicht dies eine effiziente Erkennung, welche (Sub-)Constraints bei Wertebereichseinschränkungen betroffen sind und erneut zu propagieren sind.

ten, dafür allerdings ausschließlich lineare Constraints (*linear programming*) (vgl. Borning et al. 1992, S. 246 ff.; Güsgen 2000, S. 282 f.; Van Hentenryck und Saraswat 1996, S. 711 f.).

Die Nutzung von Constraint-Hierarchien hat insbesondere in der logischen Programmierung durch **Hierarchical Constraint Logic Programming (HCLP)** eine besondere Ausprägung erfahren (vgl. Wilson und Borning 1993). Das HCLP ist eine Erweiterung des CLP-Schemas, wobei hier die Constraints innerhalb der Prolog-Regeln mit Informationen bzgl. der Constraint-Hierarchien versehen werden. Die Nutzung von Constraint-Hierarchien unterliegt allerdings auch beim HCLP denselben Einschränkungen bzgl. der existierenden Lösungsmechanismen.

Die bisher vorgestellten Verfahren zur Behandlung von OCS bewirken eine Zweiteilung der Constraints: Während ein Teil der Constraints erfüllt werden kann, ist dies für den jeweils anderen Teil nicht möglich. Ein davon abweichendes Konzept zur unscharfen Berechnung von Constraint-Problemen wird mit Fuzzy-Constraints innerhalb von **Fuzzy Constraint Satisfaction Problems (Fuzzy CSP)** verfolgt (vgl. Ruttkay 1994). Fuzzy-Constraints geben den Grad der Erfüllung einer Beschränkung für eine konkrete Wertebelegung einer Menge von Variablen an. Das heißt anstatt für eine Variable einen bestimmten Wert vorzuschreiben, wird untersucht wie weit die Belegung von dem Sollwert abweicht. Diese Abweichung wird als Wert aus dem Intervall von 0 bis 1 angegeben. Für bspw. die Variablen $v_1, \ldots, v_k$ berechnet das Fuzzy-Constraint $C(v_1, \ldots, v_k)$ für eine konkrete Belegung $(d_1, \ldots, d_k) \in D_1 \times \ldots \times D_k$ einen Wert zwischen 0 und 1. Wenn dieser Grad für $C(d_1, \ldots, d_k) = 1$ ist, bedeutet dies, dass $(d_1, \ldots, d_k)$ das Constraint C vollständig erfüllt. Wenn dagegen $C(d_1, \ldots, d_k) = 0$ ist, bedeutet dies, dass $(d_1, \ldots, d_k)$ das Constraint C vollständig verletzt. Da man bei einer Lösung für ein Fuzzy CSP nicht an einer beliebigen Wertebelegung mit einer u. U. entsprechend großen Abweichung interessiert ist, wird versucht eine Maximierung des Erfüllungsgrades für das gesamte Problem zu erreichen. Ein Fuzzy CSP ist somit ebenfalls ein Optimierungsproblem (vgl. Güsgen 2000, S. 283 f.).[18]

4.4.4 Weiterführende Konzepte

Eine gewisse Ähnlichkeit mit dem DCSP bzw. dem CompCSP weist das **Structural Constraint Satisfaction Problem (SCSP)** auf. Das SCSP erlaubt dynamisch die Struktur des Constraint-Netzes eines Problems zu verändern. Dies geschieht allerdings in diesem Fall auf der Basis von Regeln einer Graph-Grammatik (vgl. Nareyek 1999b). Dadurch lässt sich z. B. die automatische Ablaufplanung für Workflows realisieren. In einem SCSP, welches entsprechend die Generierung alternativer Pläne formalisiert, ist die Suche nach der Struktur des Constraint-Netzes demnach explizit Teil der Lösungssuche des Constraint-Erfüllungsprozesses (vgl. Nareyek 1999a).

Ein ebenfalls dynamisches Konzept, welches für offene und verteilte Umgebungen konzipiert wurde, ist das **Open Constraint Satisfaction Problem (OCSP)**. Während der klassischen Definition eines Constraint-Problems die *closed world assumption* (CWA) zugrunde liegt, d. h. es sind zu Beginn der Lösungsermittlung sämtliche Komponenten eines Problems bekannt, lassen sich durch das OCSP *open-world*-Szenarien behandeln (vgl.

[18] Zur Lösung eines Fuzzy CSP wird ebenfalls *Branch & Bound* eingesetzt.

Faltings und Macho-Gonzalez 2002).[19] Diese treten z. B. dort auf, wo offene Informationssysteme wie das Internet genutzt werden.[20] In diesem Fall werden die benötigten Informationen aus unterschiedlichen Quellen durch *Mediatoren* gebündelt bzw. übersetzt. Der Lösungsvorgang für ein OCSP gestattet es, zusätzliche Informationen, die (zusätzliche) Lösungen ermöglichen, nach und nach zu integrieren. Auf diesem Wege können Teilprobleme kombiniert werden, um eine Lösung für ein bestimmtes Problem zu erhalten. Lösungsalgorithmen für OCSPs, die mit einem *Brute-Force*-Ansatz die Informationen aller Quellen einsammeln und anschließend ein übliches (statisches) Lösungsverfahren anwenden, können sehr aufwendig werden und sind daher nur bedingt geeignet. Ein stabileres Laufzeitverhalten wird erreicht, wenn nur benötigte Informationen eingesammelt und die erste gefundene Lösung ausgeben wird. Wird festgestellt, dass mehr Informationen zur Generierung einer Lösung notwendig sind, können diese inkrementell hinzugefügt und der Lösungsprozess erneut angestoßen werden. Eine weitere Verbesserung stellt das Vorgehen dar, ausschließlich zusätzliche Informationen für bestimmte Variablen zu sammeln, die an minimalen, unlösbaren Teilproblemen partizipieren. Eine optimierte Lösung für ein OCSP wird durch „gewichtete“ Constraints erreicht, indem versucht wird, diese Gewichtungen innerhalb einer Lösung zu minimieren (vgl. Faltings und Macho-Gonzalez 2003).[21]

Ein CSP kann *verteilt* sein, weil (1) das CSP selbst logisch oder physikalisch verteilt ist, oder weil (2) die verteilte bzw. parallele Verarbeitung erhebliche Vorteile bzgl. der Effizienz mit sich bringt. Ein Konzept, bei dem der Aspekt der Verteiltheit im Vordergrund steht, ist das **Distributed Constraint Satisfaction Problem (DisCSP)**. Es wurde für die Behandlung von verteilten Abhängigkeiten in Multi-Agenten-Systemen (MAS) entwickelt. Die Variablen und Constraints eines Problems sind in einem DisCSP auf eine Vielzahl automatisierter Agenten bzw. Prozesse verteilt (engl. *inter-agent constraints*). Um Konsistenz für diese Abhängigkeiten zu gewährleisten, werden entsprechende verteilte bzw. asynchrone Lösungsverfahren auf das als DisCSP formalisierte MAS angewendet (vgl. Yokoo und Hirayama 2000).

Die nebenläufige Behandlung von Constraint-Problemen hat im **Concurrent Constraint Programming (CCP)** eine besondere Ausprägung gefunden. Das CCP ist ein Konzept für eine einfache Sprache zur effizienten und parallelen Constraint-Verarbeitung. Im Kern besteht das Konzept aus einer zentralen Kontrollinstanz zur Koordinierung, mit der eine Vielzahl Agenten bzw. Prozesse bzgl. der Problemlösung kommuniziert. Das Ziel von nebenläufiger Constraint-Verarbeitung ist es, bestehende Constraint-Probleme durch den effizienten Einsatz parallel-arbeitender Verfahren zu lösen (vgl. Frühwirth und Abdennadher 1997, S. 65 ff.; Van Hentenryck und Saraswat 1996, S. 705 f.). Nebenläufige Constraint-Behandlung nach dem CCP-Schema unterscheidet sich damit grundsätzlich im Anwendungszweck vom DisCSP und dessen Anwendung inerhalb von MAS (vgl. Yokoo und Hirayama 2000, S. 189).

---

[19] In Bezug auf die Konfigurierung wird dies auch *innovatives Konfigurieren* genannt.

[20] „Offen“ in dem Sinne, dass die Informationen und deren Menge, die theoretisch unbegrenzt sein kann, im Vorfeld nicht bekannt sind. Derartige Informationssysteme werden durch Technologien wie *Web Services* und *Semantic Web* weiter an Bedeutung zunehmen.

[21] Eine Gewichtung wird erreicht, indem jede mögliche Kombination von Variablenbelegungen eines Constraints mit bestimmten „Kosten“ markiert wird.

Weiterhin existiert ein Konzept für die Zusammenführung von verteilter und dynamischer Constraint-Erfüllung, genannt **Asynchronous Constraint Solving (ACS)**. Das Konzept für ACS ist im Rahmen einer Dissertation von *Georg Ringwelski* als sog. „Ausführungsmodell“ entwickelt worden und wird auch kurz DynDCSP genannt (vgl. Ringwelski 2003). Neben verteilten und dynamischen Aspekten unterstützt es darüber hinaus die inkrementelle Verarbeitung von Constraints, sowohl bzgl. anwachsender Constraint-Netze, als auch bezogen auf das inkrementelle Zurücknehmen von Constraints (vgl. Ringwelski 2001a; Ringwelski und Schlenker 2002).[22]

Anstatt zur Problemlösung lediglich ein einziges Lösungsverfahren zu nutzen, bietet es sich aus Effizienzgründen an, je nach Art der Problemstellung unterschiedliche Verfahren einzusetzen. Ein Konzept, welches dieses Vorgehen automatisiert, und den Constraint-Solver während des Lösungsprozesses auf das jeweilige Problem anpasst, wird durch das **Adaptive Constraint Satisfaction Problem (ACSP)** bereitgestellt (vgl. Borrett et al. 1995, 1996a, b). Mit dem ACSP wird es möglich, dynamisch während der Lösungssuche zwischen unterschiedlichen Lösungsalgorithmen zu wechseln, bis ein für das jeweilige Problem geeignetes Verfahren gefunden wird. Die Realisierung sieht vor, dass aus einer Menge vorgegebener Lösungsverfahren der Reihe nach jeweils eines ausgewählt wird. Zu Beginn kommt ein „einfaches“ Lösungsverfahren zum Einsatz, d. h. ein Verfahren, welches für einfache Problemstellungen mit durchschnittlich guter Effizienz eingesetzt werden kann. Stellt sich während der Lösungssuche heraus, dass ein Verfahren ineffizient arbeitet, wird selbiges durch ein weiteres, komplexeres Verfahren ersetzt, welches für schwieriger zu lösende bzw. härtere Problemstellungen geeignet ist. Dieses Vorgehen ermöglicht es, einfache Probleme ohne viel Overhead zu lösen, und gleichzeitig bei harten Problem die entsprechenden effizienten Verfahren nutzen zu können.

Nachteilig wirkt sich beim ACSP aus, dass der beschriebene Anpassungsprozess, bis ein für das jeweilige Problem geeigneter Algorithmus gefunden wird, u. U. relativ lange dauern kann. Zudem ist bei einem Wechsel der Suchstrategie nicht gewährleistet, dass die bisherigen Suchergebnisse übernommen werden können. Lediglich geringfügige Änderungen an der Strategie (z. B. eine Heuristik betreffend) ermöglichen i. A. die Erhaltung der bisherigen Ergebnisse. Alternativ bietet es sich daher an, die Auswahl des Lösungsverfahrens bereits zu Beginn der Lösungssuche vorzunehmen. Eine Möglichkeit, auch diesen Prozess zu automatisieren, bietet der von Kwan (1997) vorgestellte *Mapping*-Ansatz. Basierend auf der Problemstellung (zufällig erzeugte CSPs, Graph-Färbeprobleme), dem Abstand des Problems zur Phasenübergangsregion und einiger einfacher, struktureller Eigenschaften des Constraint-Netzes werden unterschiedliche Constraint-Lösungsverfahren automatisch dem jeweiligen Problem anhand von Entscheidungsbäumen, die durch ein „Maschinelles Lernverfahren“ basierend auf Fuzzy-Technologie erzeugt werden, zugeordnet.

Die Zusammenführung von finiten und infiniten Domänen innerhalb eines Problems ermöglichen Gelle und Faltings (2003) durch das **Mixed Constraint Satisfaction Problem (Mixed CSP)**. Das Konzept für Mixed CSPs kann für Problemstellungen eingesetzt werden, zu deren Behandlung es notwendig ist Constraints zu verarbeiten, die

[22] Das inkrementelle Zurücknehmen von Constraints wird von Ringwelski (2003) engl. *constraint retraction* genannt und unter dem Oberbegriff „Adaptivität“ geführt.

gleichzeitig Variablen mit finiten und infiniten Wertebereichen in Relation setzen (engl. *mixed constraints*). Derartige Constraints werden auch *heterogene Constraints* genannt (vgl. Benhamou 1996). Im Gegensatz zu Benhamou (1996), der den Konsistenzbegriff derart erweitert, dass sich diskrete und kontinuierliche Domänen gleichzeitig behandeln lassen, integrieren Gelle und Faltings (2003) bzw. Gelle (1998) unterschiedliche Lösungsalgorithmen – sowohl für diskrete Wertebereiche als auch für reellwertige Intervalldomänen – innerhalb eines Suchverfahrens zum Auffinden von Lösungen für derartig „gemischte“ Constraint-Erfüllungsprobleme. Sie definieren verschiedene Constraint-Typen für Mixed-Constraints, die z. T. unterschiedliche Möglichkeiten zur Diskretisierung von Intervalldomänen vorsehen, oder diskrete Variablen von Mixed-Constraints bevorzugt propagieren. Dies geschieht nach Möglichkeit in der Form, dass die verbleibenden, intervallwertigen Variablen ebenfalls konsistent belegt werden können. Die Autoren weisen darauf hin, dass insbesondere im Bereich der Konfigurierung Problemstellungen auftauchen können, welche die Verarbeitung von Mixed-Constraints erforderlich machen. Gelle und Faltings (2003) zeigen dafür mehrere Anwendungsbeispiele auf und erweitern das Mixed CSP durch Aktivitäts-Constraints zum *mixed CondCSP*.

Nach dieser Übersicht über bestehende Konzepte zur Constraint-Verarbeitung werden nachfolgend verfügbare Constraint-Systeme untersucht und auf ihre Eignung zur Integration in den EngCon-Konfigurator bewertet.

4.5 Verfügbare Constraint-Systeme

An dieser Stelle werden unterschiedliche Ansätze von Constraint-Systemen hinsichtlich ihrer Eignung für das strukturbasierte Konfigurierungswerkzeug EngCon betrachtet. Dieser Abschnitt untergliedert sich hierfür in vier Bereiche. Zuerst werden Programmiersprachen mit integrierten Constraint-Solvern vorgestellt, danach Bibliotheken, die Constraint-Mechanismen zur Anbindung auf programmiersprachlicher Ebene anbieten. Darauf folgend werden Constraint-Frameworks untersucht, die es ermöglichen, auf einfache Weise Constraint-Solver zu implementieren und zu nutzen. Als Letztes werden kooperative Ansätze betrachtet, die es unterschiedlichen Constraint-Solvern erlauben, Constraint-Probleme gemeinsam zu verarbeiten und aufzulösen.

Aufgrund der großen Anzahl kommerzieller und frei erhältlicher Constraint-Systeme können in diesem Abschnitt nicht sämtliche verfügbaren Constraint-Systeme im Detail aufgeführt werden. Ausführliche Übersichten und Links zu den entsprechenden Bezugsquellen sind im Internet z. B. im Archiv des *Cork Constraint Computation Centre*[23] des University College Cork, in der FAQ der Newsgroup comp.constraints[24], unter constraint.org[25], auf der Link-Seite der CSPLib[26] und im *On-Line Guide to Constraint Programming*[27] von *Roman Barták* zu finden. Im Folgenden können für jeden Teilbereich lediglich ei-

[23] `http://4c.ucc.ie/web/archive/solver.jsp`
[24] `http://www-2.cs.cmu.edu/Groups/AI/html/faqs/ai/constraints/part2/faq-doc-3.html`
[25] `http://www.constraint.org/english/tool.htm`
[26] `http://www.cse.unsw.edu.au/~tw/csplib/links.html`
[27] `http://kti.ms.mff.cuni.cz/~bartak/constraints/systems.html`

ne exemplarische Auswahl angesprochen und deren wichtigsten Eigenschaften aufgezeigt werden.

4.5.1 Integrierte Constraint-Solver

Die Allgemeinheit des CSP-Ansatzes und die Effizienz der Lösungsverfahren haben zur Entwicklung von Programmiersprachen geführt, die Constraint-Verfahren für bestimmte Problemklassen zur Verfügung stellen. Verfügbare Constraint-Systeme sind deshalb häufig Constraint-Lösungsmechanismen, die als Ergänzung innerhalb logischer Programmiersprachen basierend auf Prolog implementiert wurden. Neben spezifischen *global constraints* aus dem Bereich CP, d. h. zur Programmierung mit Constraints (vgl. Abschnitt 4.1, S. 53), werden i. d. R. ebenso allgemeine, algebraische Constraints unterstützt. Die Entwicklung dieser CLP-Sprachen begann 1982 mit **Prolog II** und führte in der zweiten Hälfte der 80er Jahre zur Entwicklung der CLP-Sprachen **CLP(R)**, **CHIP** und **Prolog III**. Während CLP(R) und Prolog III mathematische Lösungsmechanismen für linear-arithmetische Ausdrücke boten (*linear programming*), gab es in CHIP (*Constraint Handling in Prolog*) außerdem erstmals die Möglichkeit, boolesche Constraints und Constraints über benutzerdefinierte finite Domänen mit Lösungsmechanismen aus dem Bereich der Künstlichen Intelligenz innerhalb einer logischen Programmiersprache zu verarbeiten (vgl. Frühwirth und Abdennadher 1997, S. 37; Van Hentenryck 1995, S. 566). Ein moderner Prolog-Dialekt, der Lösungsmechanismen sowohl für linear-arithmetische Ausdrücke als auch für finite Domänen innerhalb eines Systems vereint, ist z. B. **SICStus Prolog** (vgl. SICStus 2004). Während diese CLP-Systeme auf die Berechnung linearer Constraints beschränkt sind,[28] wurden durch das System **CAL** (*Contrainte Avec Logique*) bzw. dessen parallel arbeitendem Nachfolger **GDCC** (*Guarded Definite Clauses with Constraints*)[29] mathematische Verfahren implementiert, mit denen sich auch nichtlineare (polynomielle) Constraints direkt lösen lassen (vgl. Frühwirth et al. 1992, S. 23; Frühwirth und Abdennadher 1997, S. 119; Hollman und Langemyr 1993, S. 113).

Ein anderes Vorgehen zur Lösung nichtlinearer Constraints wurde erstmals innerhalb von **BNR Prolog** umgesetzt: In der logischen Programmiersprache BNR Prolog kamen intervallwertige Domänen zur Repräsentation von reellwertigen Lösungen und eine entsprechende Intervallarithmetik zu deren Berechnung zum Einsatz (vgl. Frühwirth et al. 1992, S. 23; Frühwirth und Abdennadher 1997, S. 119). Moderne CLP-Systeme arbeiten häufig mit Intervalldomänen und komplexen numerisch-mathematischen Lösungsmethoden sowie Konsistenz- und Suchverfahren zur Berechnung von Constraint-Ausdrücken. Weitere Beispiele für derartige CLP-Systeme, die zusammenfassend auch **CLP(Intervals)** genannt werden, sind **CLP(BNR)**, **Interlog**, **CIAL**, **Prolog IV**, **ECL^{i}PS^e**, **DecLIC**, **CLIP**, **Newton** und **Numerica**. Diese Systeme sind zudem häufig hybrid, d. h. es lassen sich Constraints mit unterschiedlichen Wertebereichen spezifizieren und auswerten. Dies be-

[28] Ein in CLP-Sprachen wie CLP(R), Prolog III und SICStus Prolog übliches Vorgehen ist in diesem Zusammenhang das Verzögern der Auswertung nichtlinearer Constraints, bis ggf. das Constraint linear geworden ist, weil die Werte von Constraint-Variablen z. T. durch andere Constraints bereits ermittelt werden konnten (vgl. Frühwirth und Abdennadher 1997, S. 119; SICStus 2004, S. 422).

[29] GDCC basiert auf dem CCP-Paradigma (vgl. Abschnitt 4.4.4, S. 63).

trifft neben diskreten und reellwertigen Domänen ebenso intervallwertige und boolesche Wertebereiche (vgl. Benhamou und Van Hentenryck 1997, S. 107; Frühwirth und Abdennadher 1997, S. 121 f.; Van Hentenryck 1995, S. 584; Van Hentenryck und Saraswat 1996, S. 715 ff.).[30]

Eine grundsätzliche Hürde in Bezug auf die Anbindung von CLP-Systemen zur Constraint-Propagation stellt das von der Java-Architektur von EngCon abweichende, deklarative Paradigma der (constraint-)logischen Programmierung dar. Bei einer Anbindung auf programmiersprachlicher Ebene an die objektorientierte Implementierung von EngCon ist daher mit erhöhtem Integrationsaufwand zu rechnen. Zur Generierung von Constraint-Netzen in CLP-Systemen können Constraint-Gleichungen i. A. nicht direkt an einer stringbasierten Schnittstelle übergeben werden. Stattdessen wird die Erzeugung von CLP-Programmen in Prolog-Syntax in Form von logischen Klauseln notwendig. Dies kann sich negativ auf die Skalierung auswirken, wenn umfangreiche Abhängigkeiten als Constraint-Netz repräsentiert werden müssen. Zudem sind auf Prolog basierende CLP-Systeme oftmals, weil keine direkte Schnittstelle vorhanden ist, nur unter Zuhilfenahme einer Zwischenschicht (C, C++, CORBA, Sockets, etc.) in eine Java-Architektur integrierbar, was ebenso den erforderlichen Aufwand und die Komplexität der Schnittstelle erhöht.

Die für die Microsoft-Windows-Plattform mangelnde Verfügbarkeit (insbesondere frei erhältlicher Systeme) verringert die Anzahl der tatsächlich nutzbaren CLP-Systeme. Einige Systeme lassen sich z. B. durch den Einsatz der „Cygwin“-Umgebung[31] unter Microsoft Windows betreiben. Dieses erhöht allerdings abermals die Komplexität der Schnittstelle und den Integrationsaufwand.

Im Rahmen des EngCon-Projekts wurde an der Universität Bremen untersucht, wie sich der Constraint-Solver von **GNU Prolog**[32] an das Konfigurierungswerkzeug EngCon anbinden lässt. Der frei verfügbare, unter einer „Open-Source“-Lizenz stehende Prolog-Dialekt basiert auf **CLP(FD)**[33], einer CLP-Sprache für finite Domänen (vgl. Diaz und Codognet 2001). Mit GNU Prolog können daher ähnlich wie in CHIP über einen integrierten Constraint-Solver FD-Constraints ausgewertet werden. Die Ausführung und Nutzung von GNU Prolog unter Microsoft Windows ist unter Zuhilfenahme der Cygwin-Umgebung möglich. Das Resultat der Untersuchung war eine prototypische Schnittstelle, deren Kommunikation mit der Java-Applikation EngCon und auf Basis eines innerhalb von GNU Prolog implementierten Socket-Servers[34] erfolgt. Diese prototypische Umsetzung wies al-

[30] Während zur Auswertung von Constraints über boolesche Domänen bspw. in CHIP und Prolog III spezielle Constraint-Solver zum Einsatz kommen, werden in Systemen wie CLP(BNR), Prolog IV, CLP(FD) und ILOG Solver boolesche Constraints als ein spezieller Fall von FD-Constraints behandelt. Boolesche Werte werden demnach als Integer zwischen 0 (false) und 1 (true) verarbeitet (vgl. Van Hentenryck und Saraswat 1996, S. 715).

[31] Eine Linux-API-Emulationsschicht unter Microsoft Windows, welche die Kompilierung und Ausführung einer großen Anzahl Programme erlaubt, die sich mit dem frei verfügbaren GCC-Compiler kompilieren lassen: `http://www.cygwin.com`.

[32] `http://gprolog.inria.fr`

[33] `ftp://ftp.inria.fr/INRIA/Projects/contraintes/clp_fd/`

[34] `http://contraintes.inria.fr/~fages/CLPGUI/`

lerdings sowohl hinsichtlich der Funktionalität und Skalierbarkeit als auch in Bezug auf die Inkrementalität erhebliche Einschränkungen auf:

- So lassen sich mit dem in GNU Prolog integrierten Constraint-Solver, ebenso wie mit dessen Vorgänger in CLP(FD), ausschließlich finite Domänen propagieren (Wertemengen und Intervalle).[35]
- Die Implementierung als Socket-Server brachte bzgl. der Aufrufparameter Probleme bei der Variablen- und Werteübergabe mit sich.
- Ein inkrementelles Anwachsen des Constraint-Netzes im Constraint-Solver ließ sich nicht realisieren, da selbiges in GNU Prolog innerhalb einer einzigen Prolog-Klausel verwaltet wird.

Auch wenn neuere Ansätze die Anbindung existierender CLP-Systeme erleichtern (vgl. Schlenker und Ringwelski 2003), ist dennoch für jeden in einer CLP-Sprache integrierten Constraint-Solver ein Wrapper zu implementieren, für den, ähnlich wie bei der Anbindung von GNU Prolog, mit einem relativ hohem Anpassungsaufwand zu rechnen ist.[36]

4.5.2 Bibliotheken

Die Mechanismen zur Constraint-Verarbeitung sind mittlerweile so weit ausgereift, dass eine Reihe von Bibliotheken bzw. eigenständige *Constraint-Satisfaction*-Systeme erhältlich sind. Diese Systeme bzw. Werkzeuge sind dazu vorgesehen, über Programmierschnittstellen an andere Programme und Systeme angebunden zu werden. Es sind ebenfalls eine Reihe von sowohl kommerziellen als auch frei verfügbaren Systemen erhältlich. Das Leistungsspektrum der einzelnen Constraint-Bibliotheken deckt jedoch, wie nachfolgend gezeigt wird und in Tabelle 4.1 auf Seite 73 übersichtlich aufgeführt ist, jeweils nur einen Teil der für EngCon benötigten Eigenschaften ab.

So lassen sich die in Abschnitt 4.4.3 auf Seite 61 vorgestellten Constraint-Hierarchien, deren Anwendung in einem Konfigurierungssystem zweifellos interessante Möglichkeiten eröffnen würde, bedingt durch die Einschränkungen der verfügbaren Algorithmen und Constraint-Solver, nur begrenzt einsetzen. Von dem im Quellcode frei verfügbaren Constraint-Solver **Cassowary**[37] ist neben einer Version für C++ und für Smalltalk ebenso eine Java-Version erhältlich. Cassowary steht unter der *GNU Lesser General Public License*, wodurch auch eine kommerzielle Nutzung möglich ist. Allerdings ist der in Cassowary verwendete inkrementelle Simplex-Algorithmus ausschließlich in der Lage, lineare Constraints zu verarbeiten. Zudem ist weder die Propagation von Intervallen noch von finiten

[35] Für GNU Prolog ist eine Erweiterung namens CLIP verfügbar, mit der es dem Constraint-System ermöglicht wird, reellwertige Intervalle zu verarbeiten (vgl. Hickey 2000). Die Integration unterliegt allerdings, bedingt durch den Umstand, dass GNU Prolog als „Wirtssystem“ genutzt wird, denselben hier angesprochenen Einschränkungen hinsichtlich der Parameterübergabe und der Inkrementalität: `http://interval.sourceforge.net/interval/prolog/clip/`.

[36] Für das von Schlenker und Ringwelski (2003) beschriebene Framework POOC wurde zwar eine Anbindung von GNU Prolog über dessen C-Schnittstelle und das *Java Native Interface* (JNI) angestrebt (vgl. Stearns 1997, 1998), allerdings bisher nicht umgesetzt (vgl. Abschnitt 4.5.3, S. 75).

[37] `http://www.cs.washington.edu/research/constraints/cassowary/`

Domänen vorgesehen, Cassowary ist auf reellwertige Variablen beschränkt (vgl. Badros et al. 2001).

Die Firma ILOG ist seit den 90er Jahren sehr erfolgreich in Bezug auf die Entwicklung und Vermarktung von Constraint-Werkzeugen (vgl. Freuder und Wallace 2000). Der **ILOG Solver**[38] (vgl. Puget 1994; Puget und Leconte 1995) ist als Komponente innerhalb der ILOG Optimization Suite erhältlich (vgl. Abschnitt A.2, S. 268). Neben der C++-Version ist mit dem **ILOG JSolver**[39] ebenfalls eine Java-Variante verfügbar (vgl. Chun 1999a, b, c).[40] Mit den Produkten von ILOG lassen sich sowohl finite Domänen als auch reellwertige Intervalldomänen verarbeiten.[41] Zur Steigerung der Ausführungsgeschwindigkeit ist die parallele und (auf mehrere Prozessoren) verteilte Ausführung der Suchalgorithmen möglich (vgl. ILOG 2004b). Allerdings bietet der ILOG Solver nicht die Möglichkeit, ein inkrementell anwachsendes Constraint-Netz zu verwalten (vgl. Ranze et al. 2002, S. 850; Ringwelski 2002, S. 195; Ringwelski 2003, S. 24).

Ein Werkzeug mit ebenfalls hoher Funktionalität ist das kommerzielle System **UniCalc**[42]. UniCalc wurde entwickelt von der *Novosibirsk Division of the Russian Research Institute of Artificial Intelligence*. Das System basiert auf einer Kombination von Intervall-Constraint-Propagation, Intervallmathematik und Computeralgebra. Mit UniCalc lassen sich lineare und nichtlineare algebraische Ausdrücke bzw. Systeme, sowohl über die reellen Zahlen als auch ganzzahlig (*integer programming*), verarbeiten. Das Ergebnis einer Berechnung ist eine Menge von Intervallen, die alle reellen Lösungen enthalten. UniCalc ist in erster Linie ein mathematischer Problemlöser mit Schwerpunkt auf der Intervallverarbeitung. Es ist dagegen kein FD-Solver enthalten und es fehlt ebenfalls die Möglichkeit, ein inkrementell anwachsendes Constraint-Netz zu verwalten. Zudem ist UniCalc nicht in der Lage, unterbrochene Intervalle zu verarbeiten. UniCalc ist lauffähig in einer DOS/Windows-Umgebung und verfügt über eine C-Schnittstelle. Die Verfügbarkeit von UniCalc ist allerdings fraglich, da der Server des Instituts mit der offiziellen Homepage[43] aktuell online nicht erreichbar ist, und die im Internet verfügbare Demo-Version von UniCalc ein bereits beträchtliches Alter aufweist. Ob eine Weiterentwicklung bzw. Pflege des Systems erfolgt ist daher offen. Da UniCalc nicht im Quellcode vorliegt, ist ebenso keine eigene Weiterentwicklung möglich bzw. nicht durch die Open-Source-Gemeinde zu erwarten.

Ebenfalls fokussiert auf die Intervallverarbeitung ist die Bibliothek **ALIAS**[44]. ALIAS ist eine leistungsfähige Constraint-Bibliothek, die in einer Version für C++ und als Modul

---

[38] `http://www.ilog.com/products/solver/`

[39] `http://www.ilog.com/products/jsolver/`

[40] Der ILOG JSolver stammt ursprünglich nicht vom ILOG Solver, sondern von dem von *Andy Hon Wai Chun* entwickelten **JSolver** ab. Der JSolver wurde anfangs zu Ausbildungs- und Forschungszwecken frei verfügbar im Internet zum Download angeboten und war in der Lage, Constraint-Relationen über finite Domänen (Integer- und Boolesche-Variablen) zu verarbeiten (vgl. Chun 1999a, b, c). Mittlerweile wurde der JSolver in die kommerzielle Produktpalette von ILOG integriert (vgl. ILOG 2002) und ist daher nicht mehr frei im Internet zu beziehen.

[41] Durch die Erweiterung des ILOG Solver um die mathematische Bibliothek ILOG CPLEX lassen sich reellwertige Constraint-Probleme auch durch reell-mathematische Lösungsverfahren behandeln (vgl. ILOG 2004a): `http://www.ilog.com/products/cplex/`.

[42] `http://archives.math.utk.edu/software/msdos/miscellaneous/unicalc/`

[43] `http://www.rriai.org.ru/UniCalc/`

[44] `http://www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS.html`

für das Mathematik-Programm Maple erhältlich ist.[45] Sie wird seit 1999 im COPRIN-Projekt[46] des französischen INRIA[47] entwickelt (vgl. Merlet 2002). Neben Solvern zur Verarbeitung von reellwertigen Intervall-Constraints stellt ALIAS eine große Zahl weiterer intervallmathematischer Verfahren zur Verfügung. ALIAS ist Freeware für den akademischen Gebrauch und als Bibliothek sowie als ausführbares Binary unter Linux und Solaris erhältlich. Eine Live-Demo ist im Internet verfügbar.[48] Leider liegt ALIAS nicht im Quellcode vor und kann daher nicht unter Windows (z. B. mittels Cygwin) compiliert bzw. genutzt werden (vgl. COPRIN 2004).

Im Gegensatz zu ALIAS ist die C++-Bibliothek **RealPaver**[49] von *Laurent Granvilliers* im Quellcode verfügbar, da sie unter einer Open-Source-Lizenz steht. War der RealPaver vormals ebenfalls nur eingeschränkt für die wissenschaftliche Forschung verfügbar, steht er in der neuesten Version unter BSD-Lizenz, und ist dementsprechend auch kommerziell einsetzbar.[50] Die neueste Version ist dazu außer für die Betriebssysteme Linux und Solaris in einer mit Cygwin unter Windows lauffähigen Variante erhältlich. Allerdings lassen sich mit RealPaver ebenso wie mit der ALIAS-Bibliothek keine FD-Constraints behandeln. Zwar ist der RealPaver in der Lage, Integer-Werte zu verarbeiten, bietet hierfür aber keinen separaten Constraint-Solver an. Zudem werden von RealPaver weder die inkrementelle Verarbeitung von Constraints noch unterbrochene Intervalle, in Bezug auf die Wertebereiche der Constraint-Variablen, unterstützt (vgl. Granvilliers 2004).

Eine Bibliothek mit Algorithmen zur Verarbeitung von FD-Constraints ist die Bibliothek **C-Lib** von *Peter van Beek*.[51] Die Routinen dieser in der Programmiersprache C geschriebenen Bibliothek sind im Quellcode frei verfügbar (*Public Domain*) und bieten dem Entwickler Bibliotheksfunktionen für Such- und Konsistenzverfahren über finite Domänen. Allerdings ist diese Bibliothek auf FD-Constraints beschränkt und zudem ausschließlich in der Lage, binäre Tupel-Constraints anstatt implizite, intensional repräsentierte Relationen zu verarbeiten. Ein inkrementell anwachsendes Constraint-Netz ist ebenso nicht vorgesehen.

Das Java-Pendant zu van Beeks C-Bibliothek (mit einigen Erweiterungen) ist die **Java Constraint Library (JCL)**[52] von *Marc Torrens*, entstanden im Rahmen einer Diplomarbeit an der Eidgenössisch Technischen Hochschule (ETH) Lausanne[53] (vgl. Torrens 1997). Die Bibliothek ist Open-Source und steht unter LGPL. Sie kann daher ebenfalls kommerziell genutzt werden. Auch für die JCL wird im Internet eine Demonstration bereitgestellt.[54] Die JCL bietet ebenfalls eine Reihe unterschiedlicher Suchverfahren, die z. T. mit Konsistenzverfahren kombiniert werden. Selbige können allerdings auch lediglich als Preprozessing zum Einsatz kommen. Ein CSP wird zur Verarbeitung durch die JCL in einer

[45] `ftp://ftp-sop.inria.fr/coprin/ALIAS`
[46] *Constraints solving, OPtimisation, Robust INterval analysis*
[47] *Institut National de Recherche en Informatique et en Automatique*: `http://www.inria.fr`
[48] ALIAS On Line: `http://www-sop.inria.fr/coprin/aol/form.html`
[49] `http://www.sciences.univ-nantes.fr/info/perso/permanents/granvil/realpaver/`
[50] `http://sourceforge.net/projects/realpaver`
[51] `http://ai.uwaterloo.ca/~vanbeek/software/software.html`
[52] `http://liawww.epfl.ch/JCL/`
[53] *École Polytechnique Fédérale de Lausanne* (EPFL)
[54] `http://www.marctorrens.com/downloads/projects/JCL/exercise/exercise.html`

eigens implementierten Sprache, der *CSP Description Language* (CSPDL), definiert und geparst. Zur Unterstützung des Entwicklungsprozesses existiert eine grafische Oberfläche („Constraint-Shell"), in der CSPs editiert und die Lösungsprozesse initiiert werden können (vgl. Torrens et al. 1997a, b, 1998).

War die JCL in ihrer ersten Version ebenfalls ausschließlich in der Lage, extensionale Tupel-Constraints zu verarbeiten, ist sie mittlerweile um die Möglichkeit der Verarbeitung arithmetischer Constraints mit numerischen Wertebereichen, d. h. von Gleichungs- und Ungleichungssystemen über ganze Zahlen, erweitert worden. Die Entwicklung der JCL wird fortgeführt, und die nächste (stabile) Version wird voraussichtlich um die Unterstützung für SoftCSPs erweitert (vgl. Neagu et al. 2003). Jedoch unterstützt auch die JCL ausschließlich binäre FD-Constraints, d. h. es können ebenfalls weder n-äre FD-Constraints noch Intervalldomänen verarbeitet werden. Zudem bietet auch die JCL nicht die Möglichkeit zur inkrementellen Constraint-Verarbeitung.

Declarative Java (DJ)[55] stellt die Einbettung des Prolog-Dialekts B-Prolog[56] in die Programmiersprache Java dar (vgl. Zhou et al. 1998). In B-Prolog wurde dazu ein Compiler implementiert, der DJ-Programme interpretiert und das Constraint-Problem extrahiert. Anschließend wird der Constraint-Solver von B-Prolog zur Berechnung von entsprechenden Lösungen genutzt und als Ergebnis ein Java-Programm sowie eine HTML-Seite zur Visualisierung erzeugt. DJ kann daher, außer mit dem Schwerpunkt auf der Constraint-Berechnung, zur Visualisierung von Lösungen und für grafische Komponenten und GUIs, insbesondere Java-Applets in Internet-Anwendungen, eingesetzt werden. Mittels DJ lassen sich algebraische Constraints definieren, allerdings ausschließlich lineare Constraints mit finiten Domänen. Neben Wertebereichen mit finiten Wertemengen schließt dies Integer-Intervalle ein. Der Constraint-Mechanismus von DJ erlaubt kein inkrementelles Anwachsen des Constraint-Netzes. DJ ist frei verfügbar und sowohl für Windows als auch für Solaris erhältlich (vgl. Zhou 1999).

Eine weitere wie JCL vollständig in Java implementierte Constraint-Bibliothek ist der **Koalog Constraint Solver (KCS)**[57]. Der KCS der französischen Firma Koalog ist eine Bibliothek zur Programmierung mit Constraints (CP). KCS verfügt ausschließlich über fest vorgegebene FD-Constraints (*global constraints*), wie sie für spezielle Problemstellungen, benötigt werden (vgl. Abschnitt 4.1, S. 53). Der KCS kann auf programmiersprachlicher Ebene um einzelne Constraints oder Constraint-Solver erweitert werden. Eine Schnittstelle, mit deren Hilfe beliebige Constraints formuliert werden können, ein inkrementeller Constraint-Solver oder Intervallverarbeitung ist in dieser Bibliothek nicht vorgesehen (vgl. Koalog 2005).

Der Constraint-Solver **J.CP** von *Georg Ringwelski* ist eine weitere aktuelle Java Lösung (vgl. Ringwelski 2001b, 2002). J.CP ist als Prototyp im Rahmen einer Dissertation entstanden (vgl. Ringwelski 2003) und implementiert *Asynchronous Constraint Solving* (vgl. Abschnitt 4.4.4, S. 64). Mittels J.CP ist daher das verteilte Lösen von Constraints, der inkrementelle Aufbau des Constraint-Netzes sowie Constraint-Relaxierung möglich.[58]

[55] `http://www.cad.mse.kyutech.ac.jp/people/zhou/dj.html`
[56] `http://www.cad.mse.kyutech.ac.jp/people/zhou/bprolog.html`
[57] `http://www.koalog.com/php/jcs.php`
[58] Constraint-Relaxierung wird von Ringwelski (2003) engl. *constraint retraction* genannt.

Derzeit lassen sich mit J.CP ausschließlich Constraints mit Variablen über finite Domänen verarbeiten. Ringwelski (2003, S. 158) beschreibt aber, wie der Prototyp von J.CP bspw. um einen Simplex-Solver ergänzt werden kann. Allerdings ist J.CP grundsätzlich ebenso wie KCS ein System zur Programmierung mit Constraints, d. h. J.CP verfügt über eine Reihe fest implementierter Constraints. Für jedes weitere mögliche Constraint ist es erforderlich, dass jeweils eine spezielle Propagationsfunktion implementiert wird. Derartige Systeme sind für den Bereich CP geeignet, da sie die hier je nach Problemstellung erforderlichen, speziellen Constraints anbieten.[59] Im Fall von EngCon sind allerdings „generische“ Constraint-Solver für algebraische Constraints erforderlich, d. h. es muss möglich sein, anstatt einiger weniger vorgegebener Constraints, beliebige Relationen abbilden zu können.

Ein Java-basierter Constraint-Solver, der in der Lage ist, reellwertige Intervall-Constraints zu verarbeiten, ist der *Brandeis Interval Arithmetic Constraint Solver*, auch kurz **IASolver** genannt, von *Timothy J. Hickey*. Der IASolver ist als Forschungsprototyp entstanden (vgl. Hickey et al. 2000). Eine Online-Demo ist auf der Homepage des Autors verfügbar.[60] Der IASolver nutzt eine eigens in Java entwickelte Intervallarithmetik-Bibliothek namens IAMath. Sämtliche Komponenten des IASolver sind Open-Source. Sie wurden ursprünglich unter LGPL veröffentlicht, wobei die IAMath mittlerweile unter der noch weniger restriktiven „zlib/libpng“-Lizenz erhältlich ist.[61] Beide Lizenzen ermöglichen auch einen kommerziellen Einsatz der Software. Der IASolver verarbeitet ausschließlich reellwertige, kontinuierliche Intervalle und damit keine FD-Constraints. Unterbrochene Intervalle sind bisher ebenso wenig vorgesehen wie ein inkrementell anwachsendes Constraint-Netz.

In Tabelle 4.1 auf der gegenüberliegenden Seite sind die für EngCon wichtigsten Eigenschaften der vorgestellten Constraint-Bibliotheken übersichtlich dargestellt. Der ILOG Solver ist der einzige Constraint-Solver, der sowohl finite als auch infinite, d. h. reellwertige Intervalldomänen unterstützt. Rein reellwertige Wertebereiche besitzen keine Priorität für EngCon, der hierarchische Constraint-Solver Cassowary unterstützt diese ausschließlich, der ILOG Solver mit Hilfe einer Erweiterungsbibliothek. Bis auf die C-Lib lassen sich mit allen Bibliotheken Constraints in Form von intensionalen Relationen modellieren. Der KCS und J.CP fokussieren den Bereich der Programmierung mit Constraints (CP) und sind daher nicht ohne weiteres in der Lage, neben einer Reihe von vorgegebenen Constraints beliebige Relationen zu verarbeiten. Nichtlineare Constraints sind hier bezogen auf reellwertige (Intervall-)Domänen. Sie können von allen Intervallbibliotheken verarbeitet werden, FD-Solver sind nicht berücksichtigt. Ein inkrementell anwachsendes Constraint-Netz wird lediglich von ILOG JSolver, Cassowary und J.CP unterstützt. Die Erweiterbarkeit durch vorhandenen Quellcode ist bei kommerziellen Systemen naturgemäß nicht gegeben. Bibliotheken, die unter einer Open-Source-Lizenz stehen oder als *Public Domain* verfügbar sind, bieten im Gegensatz dazu diese Möglichkeit. Die APIs aller Bibliotheken sind entweder in der Programmiersprache C bzw. C++ oder in Java

[59] Wie auch CLP-Systeme, vgl. Abschnitt 4.5.1, S. 66.

[60] `http://www.cs.brandeis.edu/~tim/Applets/IAsolver.html`

[61] `http://interval.sourceforge.net/interval/java/ia_math/`

	FD	IN	RW	IS	RE	HO	NL	IK	KO	QC	SP	MS
Cassowary	–	–	x	x	x	x	–	x	–	x	Java	x
ILOG Solver	x	x	(x)	x	x	x	x	–	x	–	C++	x
ILOG JSolver	x	–	–	x	x	–	–	x	x	–	Java	x
UniCalc	–	x	–	x	x	x	x	–	x	–	C	x
ALIAS	–	x	–	x	x	x	x	–	–	–	C++	–
RealPaver	–	x	–	x	x	x	x	–	–	x	C++	x
C-Lib	x	–	–	–	x	–	–	–	–	x	C	x
JCL	x	–	–	x	x	–	–	–	–	x	Java	x
DJ	x	–	–	x	x	x	–	–	–	–	Java	x
KCS	x	–	–	x	–	x	–	–	x	–	Java	x
J.CP	x	–	–	x	–	x	–	x	–	–	Java	x
IASolver	–	x	–	x	x	x	x	–	–	x	Java	x

FD	: finite Domänen	NL	: nichtlineare Constraints
IN	: reellwertige Intervalldomänen	IK	: Inkrementalität
RW	: reellwertige Domänen	KO	: kommerzielle Bibliothek
IS	: intensionale Constraints	QC	: Quellcode verfügbar
RE	: beliebige Relationen möglich	SP	: Sprache der Schnittstelle
HO	: n-äre Constraints	MS	: für Microsoft Windows verfügbar

Tabelle 4.1: Vergleich unterschiedlicher Constraint-Bibliotheken

verfügbar.[62] Die sehr leistungsfähige Intervallbibliothek ALIAS ist leider nicht, wie für EngCon vorausgesetzt, unter dem Betriebssystem Microsoft Windows nutzbar. Für alle anderen Bibliotheken steht für diese Plattform eine Version zur Verfügung, oder sie sind durch Java plattformübergreifend einsetzbar.

Die vorgestellten Constraint-Solver decken jeder für sich genommen ein unterschiedliches Leistungsspektrum ab. Es konnte keine Bibliothek ausfindig gemacht werden, welche die Anforderungen von EngCon vollständig erfüllt. Inkrementalität wird nur von ILOG JSolver, Cassowary und J.CP unterstützt. Mit ILOG JSolver allerdings lassen sich keine Intervall-Constraints, mit Cassowary keine der erforderlichen Wertedomänen sowie keine nichtlinearen Constraints verarbeiten. J.CP hingegen ist ein Forschungsprototyp, der ausschließlich Solver für spezielle Constraints aufweist. Zudem eignen sich existierende Constraint-Solver häufig ausschließlich für bestimmte Problemstellungen, insbesondere betrifft dies die unterstützten Wertebereiche. Bis auf den ILOG Solver unterstützt keine Bibliothek sowohl finite Wertebereiche als auch Intervalldomänen.

Zur Unterstützung unterschiedlicher Domänen ist eine Kombination von Constraint-Solvern notwendig. Um Constraint-Solver mit unterschiedlichen Eigenschaften für eine Anbindung an EngCon zu kombinieren ist eine einheitliche Schnittstelle für deren Integration bzw. ein Framework erforderlich, in dem diese Solver implementiert werden kön-

[62] Systeme mit einer Schnittstelle in C/C++ können per JNI an an EngCon angebunden werden (vgl. Stearns 1997, 1998).

nen. Im Folgenden Abschnitt werden einige Ansätze für derartige Constraint-Frameworks vorgestellt.

4.5.3 Frameworks

Von Roy et al. (2000) wird ein Constraint-Framework als Möglichkeit beschrieben, eine flexible und erweiterbare Implementierung des Constraint-Mechanismus anzubieten. Anstatt wie bei CP- bzw. CLP-Systemen eine Domäne, nämlich die der (logischen) Programmierung mit Constraints zu fokussieren, bietet ein Constraint-Framework allgemeine Mechanismen, die eine Anpassung für spezielle Problemstellungen erlauben. In einem Framework werden demnach kollaborierende Komponenten integriert, die auf diese Weise eine wiederverwendbare Architektur für eine Vielzahl von Anwendungen zur Verfügung stellen. Weil ein Framework üblicherweise bereits funktionsfähige Mechanismen beinhaltet, kann es häufig auch *out of the box* wie eine Bibliothek eingesetzt werden.

In dem in der Programmiersprache Smalltalk implementierten Constraint-Framework **BackTalk** (*Backtracking in Smalltalk*) erfolgt die konsequente Ausnutzung der Eigenschaften der objektorientierten Entwicklung bzw. Programmierung (Vererbung, Polymorphie, etc.). BackTalk erlaubt die intensionale Definition von Constraints mit finiten Domänen und stellt zu deren Auflösung eine Reihe unterschiedlicher Such- und Konsistenzalgorithmen zur Verfügung. Constraints sind z. T. vorgegeben, können aber auch in einer einfachen Syntax algebraisch als Gleichung bzw. Ungleichung angegeben werden (auch n-äre Constraints). Die Erweiterbarkeit von BackTalk wird gewährleistet durch eine objektorientierte Schnittstelle, an der durch Vererbungsmechanismen sowohl neue Constraints als auch Lösungsverfahren integriert werden können. Zusätzliche Such- und Konsistenzalgorithmen werden in das Framework eingebunden, indem innerhalb einer Klassenhierarchie die Default-Lösungsverfahren einer übergeordneten Kontrollschleife von spezialisierten Unterklassen überschrieben werden können. Die Mechanismen der Vererbung werden hier genutzt, um Anpassungen an spezielle Problemstellungen vornehmen zu können. Lösungsalgorithmen sind demnach eine Bibliothek von austauschbaren Smalltalk-Klassen (vgl. Roy et al. 1999; Roy und Pachet 1997; Roy et al. 1997).

Im Gegensatz zu BackTalk implementiert **GIFT** (*Generic Interface for FilTers*) kein eigenes Constraint-System, sondern bietet die Möglichkeit, Algorithmen zur Constraint-Propagation über eine einheitliche C++-Schnittstelle verfügbar zu machen (vgl. Ka Boon et al. 2000). Durch diese allgemeine Schnittstelle wird eine Wiederverwendung existierender Lösungsalgorithmen in unterschiedlichen Constraint-Systemen erleichtert, die ansonsten für jedes spezielle System reimplementiert werden müssten. Anstatt Lösungsalgorithmen direkt für ein bestimmtes System zu implementieren, muss für jedes spezielle Constraint-System lediglich eine C++-Schnittstelle für GIFT erstellt werden, welche den Zugriff auf unterschiedliche Lösungsverfahren erlaubt. Allerdings ist GIFT kein Framework im eigentlichen Sinn, sondern lediglich eine Schnittstelle zur Vereinheitlichung des Zugriffs auf Constraint-Lösungsalgorithmen. Zudem liegt der Fokus wiederum auf Systemen zur Programmierung mit Constraints, d h. GIFT ist darauf beschränkt, den Zugriff auf spezielle Constraints für CP- bzw. CLP-Systeme zu vereinheitlichen.

Wie GIFT ist **POOC** aus dem Gedanken heraus entstanden, eine einheitliche Schnittstelle für den Zugriff auf bestehende Constraint-Lösungsverfahren anzubieten (vgl. Schlenker und Ringwelski 2003). Im Fall von POOC geschieht dies allerdings nicht für spezielle C(L)P-Systeme, sondern allgemein für die Programmiersprache Java.[63] Über eine objektorientierte Schnittstelle werden von POOC wie in BackTalk insbesondere Vererbungsmechanismen genutzt, um in einer Java-Umgebung den einheitlichen Zugriff auf unterschiedliche Constraint-Solver zu gewährleisten. Bestehende Constraint-Systeme können über einen Wrapper an POOC angebunden werden. Ein einfacher Solver namens *firstcs* ist in POOC enthalten (vgl. Hoche et al. 2003). Für SICStus Prolog existiert bereits eine Wrapper-Klasse. Die Anbindung von GNU Prolog wurde nicht fertig gestellt, aber konzeptionell über das JNI dargelegt (vgl. Schlenker und Ringwelski 2003, S. 160 u. 169). Der Schwerpunkt von POOC liegt ebenfalls auf der (objektorientierten) Programmierung mit Constraints (CP). POOC ist daher entsprechend optimiert auf die Verarbeitung spezieller Constraint-Typen (*global constraints*), wobei auch die Verarbeitung n-ärer Constraints möglich ist. Die Wertebereiche von Constraint-Variablen sind allerdings auf finite Domänen beschränkt.

Kein Framework im objektorientierten Sinn, aber ebenfalls ein Rahmenwerk zur Constraint-Verarbeitung sind die **Constraint Handling Rules (CHR)** von *Thom Frühwirth*.[64] CHR sind ein flexibles Mittel zur Implementierung von benutzerdefinierten Constraints bzw. Constraint-Solvern.[65] Sie sind eine Spracherweiterung, die als Bibliothek in existierende Programmiersprachen wie Prolog, Lisp, Haskell und Java eingebunden werden können. Umfangreiche Implementierungen von CHR-Bibliotheken werden für SICStus Prolog (Referenzimplementierung, vgl. SICStus 2004, S. 493 ff.) und ECL^{i}PS^e (vgl. Frühwirth 1998, S. 112 ff.) angeboten.[66] In der Basissprache implementierte Programme sind in der Lage, die in CHR implementierten Constraints zu nutzen. Umgekehrt lassen sich von CHR-Programmen in der Basissprache implementierte Prozeduren als Hilfsprozeduren verwenden. CHR stellen dabei eine höhere deklarative Sprache zur Spezifikation und Implementierung von Constraint-Lösungsmechanismen dar. Dies geschieht innerhalb von CHR-Programmen in Form von Regeln, mit denen beschrieben wird, wie sich Constraints zu neuen Constraints vereinfachen lassen und wie durch Constraints andere Constraints propagiert werden. Fortlaufende Vereinfachung bzw. Propagierung führt letztendlich zu einer Lösung der Constraints (vgl. Frühwirth 1995; Frühwirth und Abdennadher 1997, S. 78 ff.).

CHR sind aus der Problemstellung entwickelt worden, dass unterschiedliche Klassen von Anwendungen im Bereich CP oftmals unterschiedliche Constraints benötigen.[67] Neuartige Constraints, bzw. neue *global constraints*, können u. U. nur mit viel Aufwand in existierende, einfachere Constraints übersetzt werden. Zur Kompensation dieser Problematik lassen sich mit CHR auf hohem Abstraktionsniveau Constraint-Solver für spezi-

[63] POOC wird von J.CP (vgl. Abschnitt 4.5.2, S. 71) eingesetzt (vgl. Ringwelski 2003, S. 37 u. S. 80).

[64] `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`

[65] In diesem Fall auch Constraint-*Handler* genannt.

[66] WebCHR Online-Demo: `http://bach.informatik.uni-ulm.de/~webchr/`

[67] Mit „Constraint-Solver“ sind im Zusammenhang mit CHR i. d. R. Constraint-Solver bzw. Propagationsfunktionen für spezielle Constraints für den CP-Bereich gemeint.

fische Constraints innerhalb von bestehenden C(L)P-Systemen implementieren. Neben einer Vielzahl spezieller Constraints werden von den vorhandenen CHR-Bibliotheken auch Constraint-Solver mit allgemeinen Such- und Konsistenzverfahren zur Verfügung gestellt (vgl. Frühwirth 1998, S. 26 ff.).

Neben CHR-Bibliotheken für logische und funktionale Programmiersprachen ist mit **JACK** (*Java Constraint Kit*)[68] eine CHR-Implementierung für die objektorientierte Sprache Java verfügbar. JACK enthält neben der CHR-Implementierung JCHR ein Framework JASE (*Java Abstract Search Engine*) zur die Einbindung unterschiedlicher Constraint-Lösungsalgorithmen, sowie die Komponente VisualCHR zur Visualisierung der Vereinfachungs- und Propagationsschritte (vgl. Abdennadher et al. 2001, 2002a, b). Die Funktionalität von JACK ist allerdings im Vergleich zu anderen CHR-Bibliotheken sehr beschränkt, da es sich bei der einzig verfügbaren und bereits im Jahr 2001 veröffentlichten Version lediglich um ein Pre-Release handelt. Im Wesentlichen wird von JACK die Programmierung von Constraints mit finiten Domänen unterstützt. Die Komponente JASE bietet allerdings beim derzeitigen Versionsstand auch hier lediglich einfache Suchstrategien für ausschließlich binäre Constraints an.[69] Die eingeschränkte Funktionalität erklärt sich z. T. mit den fehlenden vorhandenen Möglichkeiten zur Constraint-Verarbeitung in Java, und zum anderen mit dem derzeit prototypischen Status der Implementierung von JACK.[70]

Zusammenfassend kann festgestellt werden, dass keiner der bestehenden Framework-Ansätze für eine Integration in das Konfigurierungswerkzeug EngCon in Frage kommt. Das Smalltalk-Framework BackTalk geht konzeptionell in die richtige Richtung, ist allerdings auf finite Domänen beschränkt. Einer Nutzung im Rahmen von EngCon steht zudem die Sprachbarriere im Weg (Smalltalk ↔ Java). Im Gegensatz zu BackTalk ist GIFT kein Constraint-System, sondern eine reine Schnittstelle zur Vereinheitlichung des Zugriffs auf existierende Constraint-Lösungsverfahren. Das POOC-Framework ist ebenfalls auf finite Domänen beschränkt und fokussiert zudem den Bereich CP. In POOC enthalten ist lediglich ein einfacher Constraint-Solver. Bei CHR steht die Implementierung spezieller Constraints im Vordergrund, wodurch ebenfalls der Bereich C(L)P, im Fokus steht. Leistungsfähige CHR-Bibliotheken benötigen ein in Bezug auf die Constraint-Verarbeitung ebenso leistungsfähiges „Wirtssystem" wie SICStus Prolog oder ECL^{i}PS^e. Für eine mögliche Integration dieser Systeme in EngCon ist mit Problemen und erhöhtem Aufwand wie in Abschnitt 4.5.1 auf Seite 66 ff. beschrieben zu rechnen. Zudem sind derartige Systeme i. d. R. im Gegensatz zu CHR nicht frei verfügbar. Für die Java-basierte CHR-Implementierung JACK würde der Integrationsaufwand geringer ausfallen, allerdings weist JACK mit seinem bisherigen Stand als Forschungsprototyp lediglich eine stark eingeschränkte Funktionalität auf.

---

[68] `http://www.pms.ifi.lmu.de/software/jack/`

[69] Die von Abdennadher et al. (2001, 2002a, b) angedeutete Intervallfunktionalität beschränkt sich auf ein Constraint zur Einschränkung des Wertebereichs einer FD-Variable innerhalb zu definierender Grenzen. Ähnlich eingeschränkt verhält es sich bei der Verarbeitung reellwertiger Constraints, die linear sein und in Polynomialform vorliegen müssen.

[70] Dem Vorteil der deklarativen Erstellung bzw. Erweiterung von Constraint-Solvern in CHR steht der Aufwand gegenüber, dass bzgl. einer Weiterentwicklung von JACK die hierfür benötigten Grundlagen erst in Java geschaffen, d. h. implementiert, werden müssen.

4.5.4 Kooperative Ansätze

Die Kooperation unterschiedlicher Constraint-Solver zur gemeinsamen Verarbeitung und Auflösung von Constraint-Problemen wurde in der Vergangenheit mehrfach untersucht (vgl. Granvilliers et al. 2001a, b, c). Die vorliegenden Arbeiten lassen sich grundsätzlich unterscheiden in Ansätze, die eine statische Kooperation mehrerer Constraint-Lösungsverfahren vorsehen, und Ansätze, die sich flexibel hinsichtlich der einzusetzenden Verfahren und z. T. auch hinsichtlich der Art und Weise der Kooperation verhalten.

4.5.4.1 Ad-hoc-Kooperationen

Ansätze, die eine statische Kooperation vorsehen, auch *Ad-hoc-Kooperationen* genannt, sind in der Anwendung beschränkt auf eine bestimmte Domäne, fest vorgegebene Constraint-Solver und/oder eine bestimmte Lösungsstrategie, wie die Constraint-Solver zusammenwirken (vgl. Granvilliers et al. 2001a, b, S. 3; Granvilliers et al. 2001c, S. 159 ff.; Monfroy 1998, S. 349; Monfroy 2000, S. 212). Beispiele für derartige Ansätze im Bereich der infiniten Domänen sind die von Benhamou und Granvilliers (1996) und Granvilliers (2001) vorgestellten Kooperationen. Benhamou und Granvilliers (1996) kombinieren algebraische Berechnungsmethoden mit Konsistenzverfahren für intervallwertige Domänen, Granvilliers (2001) unterschiedliche Algorithmen zur Konsistenzherstellung auf Intervalldomänen. Ziel ist in diesen Fällen die effizientere Verarbeitung der Problemstellungen. Auch kombinierte Lösungsalgorithmen für finite Domänen, wie sie z. B. in der Arbeit von Prosser (1993b) vorgestellt werden, können als statische Kooperationen aufgefasst werden.

Marti und Rueher (1995) und Rueher (1995) kombinieren (reellwertige) algebraische und intervallarithmetische Lösungsverfahren. Sie entwickeln ein verteiltes System mit asynchron kommunizierenden Agenten, basierend auf dem Paradigma des *Concurrent Constraint Programming* (CCP). Das System ermöglicht es Probleme zu lösen, die ein einzelner der eingesetzten Constraint-Solver nicht bewältigen könnte.

Kooperationen von Solvern, die Constraints mit vermischten Domänen verarbeiten können, sind die bereits genannten Ansätze zur Verarbeitung von Mixed CSPs (vgl. Benhamou 1996; Gelle 1998; Gelle und Faltings 2003). Sie bieten die Möglichkeit Constraints zu verarbeiten, die Variablen sowohl mit finiten als auch infiniten Domänen beschränken, sind allerdings ebenfalls statische Ad-hoc-Kooperationen bzgl. der eingesetzten Lösungsverfahren und der Lösungsstrategie.

Constraint-Solver mit einer Ad-hoc-Kooperationen lassen sich z. T. auf Implementierungsebene in Form von kombinierten Algorithmen zügig erstellen und einsetzen. Auch eine Verarbeitung von unterschiedlichen Domänen ist möglich. Der ausschließliche Einsatz von Ad-hoc-Kooperationen ist jedoch grundsätzlich weniger geeignet, ein flexibles System vor dem Hintergrund einer effizienten Verarbeitung unterschiedlicher Problemstellungen zu erstellen.

4.5.4.2 Flexible Kooperationen

In ihrer Dissertation stellt *Petra Hofstedt* ein allgemeines und formales Schema für die Kombination von Constraint-Systemen und die Kooperation von Constraint-Solvern zur

Behandlung von Constraints mit vermischten Wertedomänen vor (vgl. Hofstedt 2001). Strategien werden hier in Form von frei modellierbaren *Kooperationsstrategien* eingesetzt. Mit diesen kann beschrieben werden, welche Constraint-Verfahren wie und in welcher Reihenfolge (sequentiell/parallel) kombiniert werden. Ebenfalls ist in diesem Ansatz ein Meta-Constraint-Solver vorgesehen, der allerdings im Wesentlichen zur Verwaltung des Constraint-Netzes und zur Koordinierung der einzelnen Constraint-Solver eingesetzt wird (vgl. Hofstedt 2000). Das theoretische Schema von Hofstedt (2000, 2001) wurde im Rahmen einer Diplomarbeit prototypisch in der Programmiersprache Java umgesetzt (vgl. Godehardt und Seifert 2001). Als Constraint-Solver kommen der IASolver für reellwertige Intervalle (vgl. Abschnitt 4.5.2, S. 72), eine Reimplementierung der C-Lib für finite Domänen (vgl. Abschnitt 4.5.2, S. 70) und ein eigenimplementierter Simplex-Solver für linear-arithmetische Constraints zum Einsatz (vgl. Hofstedt et al. 2001, S. 13 ff.). Das von Hofstedt (2000, 2001) vorgestellte formale Schema ist sehr flexibel, allerdings auch sehr komplex, so dass die Implementierung dagegen recht einfach gehalten ist und nur eine einzige, statische Kooperationsstrategie enthält (vgl. Hofstedt et al. 2001, S. 16). Neuere Arbeiten zielen daher darauf ab, mögliche Strategien mittels einer Beschreibungssprache (engl. *strategy description language*) dynamisch spezifizieren und flexibel einsetzen zu können. Ein Prototyp ist bisher allerdings ausschließlich für Common Lisp verfügbar (vgl. Frank et al. 2003a, b).

Eric Monfroy widmet sich in seiner Dissertation und in seiner Habilitation ebenfalls dem Themenbereich kooperierender Constraint-Solver (vgl. Monfroy 1996, 2002). Er entwickelt eine Sprache namens BALI zur Steuerung von Kooperationen. BALI erlaubt es, die Kooperation unterschiedlicher Constraint-Solver auf hoher Ebene durch eine eigene Sprache zu beschreiben, und auf diese Weise effizient neue Prototypen von kooperierenden Solvern zu erstellen (vgl. Monfroy 1998, 2000). Neben unterschiedlichen Kooperationsprimitiven, welche die sequentielle, die unabhängig parallele und die nebenläufige Ausführung von Constraint-Lösungsmechanismen erlauben, bietet BALI mehrere Lösungsstrategien an. Darunter befindet auch sich eine inkrementelle Variante (vgl. Monfroy 2000, S. 215 u. S. 224). Die Implementierung der Kooperationssprache BALI wurde innerhalb des CLP-Systems ECL^{i}PS^e[71] realisiert, welches dementsprechend als „Host-Sprache“ fungiert (vgl. Monfroy 1996, S. 163).

Ein neuerer Ansatz besteht in der Nutzung von speziellen Koordinierungssprachen zur Steuerung der Kooperation von Constraint-Solvern. Eine Reimplementierung von BALI wurde innerhalb der Koordinierungssprache Manifold[72] umgesetzt (vgl. Arbab und Monfroy 1998a, b). Ein weiterer Prototyp, der ebenfalls auf Manifold aufsetzt, ist DICE (*DIstributed Constraint Environment*), in dem allerdings, im Gegensatz zu BALI und dem System von Hofstedt (2000), nur sehr begrenzte Möglichkeiten hinsichtlich einer flexiblen Kooperation vorhanden sind (Zoeteweij 2002, 2003). Koordinierungssprachen unterstützen die Interaktionen und das kooperative Zusammenwirken mehrerer Komponenten. Sie erlauben die saubere Trennung zwischen den eigentlichen Berechnungen eines Systems und der Koordinierung von Kooperationen (vgl. Arbab 1998a, b). Nachteilig wirkt sich

[71] `http://www.icparc.ic.ac.uk/eclipse/`

[72] `http://www.cwi.nl/projects/manifold/manifold.html`

der Aufwand für die Koordinierung der Kooperation aus (vgl. Zoeteweij 2003, S. 183; Ringwelski 2003, S. 3). Dieser zusätzliche Aufwand muss durch die gemeinsame bzw. verteilte Berechnung unterschiedlicher Constraint-Solver kompensiert werden, um trotzdem einen Effizienzgewinn zu erreichen. Derartige Systeme lassen sich folglich nur bei umfangreichen Constraint-Problemen effizient einsetzen.

4.6 Fazit

Bestehende Werkzeuge zur Behandlung von Constraint-Problemen erfüllen nicht die konkreten Anforderungen bzgl. einer Integration in das Konfigurierungswerkzeug EngCon. Es konnte kein System ermittelt werden, welches zusammen die relevantesten Forderungen erfüllt, d. h. es muss „hybrid" sein und damit sowohl finite Domänen als auch reellwertige Intervalle unterstützen, beliebige Relationen in Form von algebraischen Constraints sowie den inkrementellen Aufbau des Constraint-Netzes ermöglichen und von der Programmiersprache Java aus unter Microsoft Windows nutzbar sein. Eine einfache, stringbasierte Schnittstelle wird außer vom IASolver von keinem System angeboten.

Auch wenn im Bereich CLP durchaus hybride Systeme existieren (z. B. BNR Prolog, CLP(BNR), ECL^{i}PS^e), besteht das Problem, dass sich diese Systeme i. d. R. nur schwer adaptieren lassen, da eine Java-Schnittstelle häufig nicht enthalten ist, bzw. weil diese Systeme für die (logische) Programmierung mit Constraints ausgelegt sind.[73] Damit wird zwar üblicherweise eine Schnittstelle zur Nutzung von in diesem Kontext benötigten *global constraints* angeboten, aber keine einfach zu nutzende (stringbasierte) Schnittstelle für beliebige Relationen. Für die Integration in EngCon sind allerdings generische Constraint-Solver zur Auflösung derartiger Constraints erforderlich, anstatt generische, d. h. wiederverwendbare, Constraints aus dem CP-Bereich.

Um dies zu ermöglichen scheint eine Eigenentwicklung am Ehesten geeignet. Die Erfahrungen hinsichtlich der eingeschränkten Funktionalität und Skalierbarkeit bei der prototypischen Integration von GNU Prolog in EngCon bestärken dies: Der Aufwand für die komplexe Integration eines Fremdsystems gestaltet sich u. U. größer, als die Implementierung eigener Algorithmen zur Constraint-Verarbeitung. Vorausgesetzt das *Know-How* bzgl. der entsprechenden Constraint-Lösungsverfahren ist vorhanden (vgl. Kapitel 5).

Eine Schnittstelle zur Anbindung an EngCon muss in jedem Fall entwickelt und implementiert werden. Die durch diese Schnittstelle bereitgestellte Umgebung ist maßgeblich für die Integration von Constraint-Verfahren in EngCon, seien es Fremdsysteme oder Eigenentwicklungen. Der Nutzen einer möglicherweise komplizierten Anbindung eines Fremdsystems an diese Schnittstelle kann sich dagegen im Vergleich zur Eigenimplementierung von Lösungsalgorithmen als recht gering erweisen, besonders wenn die im Fremdsystem vorhandenen Constraint-Lösungsverfahren nicht sehr anspruchsvoll sind (siehe z. B. JACK oder *firstcs* innerhalb von POOC). Dabei würde von einem Fremdsystem ausschließlich der Zugriff auf dessen Constraint-Verarbeitung benötigt. Relevant für EngCon ist nicht

[73]Zudem sind CLP-Systeme z. T. nur unter Unix-Systemen nutzbar, und nicht wie erforderlich unter Microsoft Windows. Ob diese Systeme in einer Umgebung wie Cygwin in der erforderlichen Stabilität nutzbar sind (vorausgesetzt der Quellcode für den notwendigen Kompilierungsvorgang ist verfügbar), kann nicht garantiert werden, wenn nicht entsprechende Pakete verfügbar sind.

die Deklarativität z. B. einer Prolog-Umgebung oder des CHR-Frameworks, sondern der stringbasierte Zugriff auf deren Constraint-Lösungsverfahren und die möglichst flexible Einbettung in EngCon.[74]

Eine Eigenimplementierung für EngCon hätte außerdem den Vorteil der Erweiterbarkeit des Constraint-Systems, z. B. hinsichtlich Constraint-Hierarchien oder Constraint-Relaxierung. Der verfügbare Quellcode würde die Plattformunabhängigkeit bzw. die Möglichkeit der Portierung des Systems sicherstellen. Zudem kann bei einer Eigenentwicklung dafür Sorge getragen werden, dass die Anforderungen von EngCon erfüllt werden, soweit dies entsprechend dem Aufwand und der verfügbaren Kapazitäten möglich ist.

Um je nach Problemstellung flexibel geeignete Lösungsverfahren einsetzen zu können, bietet sich für die zu entwickelnde Constraint-Komponente ein objektorientierter Framework-Ansatz an. Da für EngCon ein hybrides Constraint-System, sowohl für finite als auch infinite Constraint-Domänen benötigt wird, sind in jedem Fall unterschiedliche Constraint-Solver erforderlich. Neben eigenen implementierten Constraint-Solvern können in ein derartiges Framework über einen Wrapper-Mechanismus zudem bestehende Constraint-Systeme eingebunden werden, die, wenn sie die Anforderungen von EngCon auch nicht vollständig erfüllen, für bestimmte Problemstellungen ausreichend (oder gar notwendig) sind. Ein Framework bietet die notwendige Umgebung, um relativ einfach neue Lösungsverfahren direkt implementieren bzw. Fremdsysteme integrieren zu können. Es bietet weiterhin den Vorteil der modularen Erweiterbarkeit und allgemeine Schnittstellen für den flexiblen Einsatz in unterschiedlichen Anwendungen auch außerhalb von EngCon.

Zur Verarbeitung von Constraints mit finiten und infiniten Domänen müssen von den unter Abschnitt 4.4 auf Seite 57 ff. angesprochenen Ansätzen für eine Eigenentwicklung in jedem Fall das klassische CSP und das ICSP berücksichtigt werden. Da innerhalb eines hybriden Systems grundsätzlich auch heterogene Constraints auftreten können, ist das Mixed CSP ebenfalls als ein sinnvolles Konzept bezgl. einer Eigenentwicklung anzusehen. Die genannten übergeordneten Constraint-Verfahren zur Konfigurierung sind weniger relevant, da für das existierende, *interne* Constraint-System in EngCon dynamische Mechanismen durch die inkrementelle Instantiierung von Constraint-Relationen durch den Pattern-Matcher bereits vorgesehen sind. Die weitergehend angesprochenen Verfahren zur Behandlung von überbestimmten Constraint-Problemen sind insbesondere für ein interaktives Konfigurierungssystem wie EngCon sehr interessant, müssten aber ebenfalls konzeptionell übergeordnet behandelt werden. Da dies massive Eingriffe in das interne Constraint-System von EngCon beinhalten würde, die über den Austausch des Constraint-Solvers bzw. der Erstellung einer Schnittstelle oder Frameworks hinausgehen, kommt eine Bearbeitung innerhalb dieser Arbeit nicht in Frage. Die genannten weiterführenden Konzepte, insbesondere das OCSP[75] und ACSP, sind in Bezug auf EngCon

[74] Eine deklarative Prolog- oder CHR-Schnittstelle zur Programmierung mit Constraints (CP) ist für den konkreten Einsatz in EngCon grundsätzlich eher ungeeignet und würde eine Nutzung des entsprechenden Constraint-Systems durch den entstehenden Overhead aufwendiger machen.

[75] Das Konzept für ein OCSP ist in der derzeitigen Version von EngCon nicht anwendbar, weil dieser die *closed world assumption* (CWA) zugrunde liegt. Im Rahmen der sog. *innovativen Konfigurierung* wäre es allerdings z. B. in einer zukünftigen, verteilten EJB-Umgebung denkbar, dass unterschiedliche Datenquellen (Wissensbasen) unter Voraussetzung der *open-world*-Annahme genutzt werden können.

ebenfalls sehr interessant, eine Bearbeitung würde allerdings auch hier über den Rahmen dieser Arbeit hinausgehen. Für eine Integration von übergeordneten und weiterführenden Konzepten zur Constraint-Verarbeitung müsste eine ausführliche Evaluierung erfolgen, in der geklärt wird, inwieweit diese Konzepte für den Einsatz in EngCon bzw. innerhalb der Konfigurierung allgemein geeignet sind.

In dem nachfolgenden Kapitel 5 werden die für eine hybride Constraint-Verarbeitung in EngCon benötigten Verfahren für klassische CSPs und ICSPs im Detail vorgestellt. Auf das Mixed CSP zur Verarbeitung von heterogenen Constraints wird in Kapitel 6 bei der Vorstellung des Konzepts für ein hybrides Constraint-System eingegangen.

Kapitel 5

Grundlegende Constraint-Lösungstechniken für ein hybrides System

Der Mangel an mathematischer Bildung
gibt sich durch nichts so auffallend zu erkennen,
wie durch maßlose Schärfe im Zahlenrechnen.

Carl Friedrich Gauss

Im ersten Teil dieses Kapitels werden Konsistenz- und Suchverfahren zum Propagieren und Lösen von klassischen Constraint-Problemen mit finiten Domänen behandelt. Dies beinhaltet Heuristiken zur Optimierung der Lösungssuche. Neben den Lösungsverfahren für binäre Constraint-Probleme werden außerdem Verfahren für n-stellige Probleme vorgestellt. Im zweiten Teil des Kapitels wird ein Überblick über Lösungsverfahren für intervallwertige Constraint-Probleme gegeben. Neben intervallarithmetischen Grundlagen beinhaltet dies Splitting-Techniken und Konsistenzalgorithmen, angewendet auf Intervalldomänen.

5.1 Einführung

Constraints über finite Domänen sind ein klassisches Gebiet der KI. Sie kommen in vielen Anwendungen zum Einsatz, die Problemstellungen mit endlichen Wertebereichen beinhalten. Während es zur Lösung von Constraints über finite Domänen bereits eine Reihe von effizienten Standardverfahren gibt, auf die in Abschnitt 5.2 eingegangen wird, haben die infiniten Domänen bisher deutlich weniger Beachtung gefunden. Zwar gibt es sowohl für lineare als auch für nichtlineare Gleichungssysteme mit kontinuierlichen Domänen bekannte mathematische Verfahren zum Auflösen der Constraints, die z. B. in verschiedenen kommerziellen Constraint-Systemen zum Einsatz kommen (vgl. Benhamou 1995, S. 17; Van Hentenryck 1989, S. 202), allerdings sind diese Verfahren jeweils auf spezielle algebraische Ausdrücke begrenzt (vgl. Sam-Haroud 1995, S. 15 ff.): So wird für linear-arithmetische Ausdrücke die Gauß-Jordan-Eliminierung (Gleichungen) bzw. der Simplex-Algorithmus (Ungleichungen) eingesetzt (vgl. Bronstein et al. 1996, S. 1099 ff. u. S. 1002 ff.). Nichtlineare algebraische Gleichungssysteme können mit Hilfe des Buchberger-Algorithmus (Gleichungen) und des CAD-Algorithmus[1] (Ungleichungen) aufgelöst werden (vgl. Hollman und Langemyr 1993), wobei letztere auf polynomielle Constraints beschränkt sind. Es ist bisher kein rechnergestütztes, *algebraisches* Verfahren bekannt, mit dem allgemeine, nichtlineare Constraint-Systeme gelöst werden könnten. Die meisten *numerischen* Algorithmen hingegen garantieren weder Vollständigkeit noch Korrektheit. So können mögliche Lösungen „verloren gehen", ein globales Optimum wird u. U. nicht gefunden und manchmal konvergieren numerische Verfahren schlicht nicht (vgl. Lebbah und Lhomme 2002, S. 110). Numerische Algorithmen, die diese Eigenschaften erfüllen, stammen entweder aus der Intervallanalysis oder aus dem Bereich der KI und werden in Abschnitt 5.3 vorgestellt.

5.2 Klassische Constraint Satisfaction Probleme

Das allgemeine *Constraint Satisfaction Problem* (CSP) bezeichnet eine Klasse von kombinatorischen Problemen, die mittels einer Menge von Randbedingungen bzw. Constraints über eine Menge von Variablen formuliert werden. Die Aufgabe besteht darin, eine wohlgeformte Belegung für eine endliche Menge von Variablen zu finden. Wohlgeformt bedeutet in diesem Fall eine konsistente Belegung der Variablen mit Werten, so dass alle Randbedingungen erfüllt werden.

Das CSP-Schema ist ein flexibler Ansatz zur natürlichen Problembeschreibung durch Constraints. Der Einsatz von Constraints erlaubt eine formale und deklarative Problembeschreibung, die von konkreten, domänenspezifischen Lösungsverfahren abstrahiert. Probleme werden als CSP formuliert, weil dadurch die Beantwortung bestimmter Fragestellungen durch spezielle Lösungsverfahren auf effiziente Weise ermöglicht wird (vgl. Kolbe 2000, S. 37).

Die Ursprünge des CSP gehen zurück auf die Arbeit von Montanari (1974). Erstmals wurde ein reales Problem einer Anwendung von Waltz (1975) als CSP behandelt. Eine Erweiterung bzw. ein erster Vergleich verschiedener Lösungsverfahren wurde von Mackworth (1977a) vorgenommen. Eine gute Übersicht ist dem Lehrbuch von Tsang (1993) und den Übersichtsartikeln bzw. -kapiteln von Barták (1999a, b, 2001); Dechter (1992, 1999); Dechter und Rossi (2003); Güsgen (2000); Kumar (1992); Russel und Norvig (2002); Ruttkay (1998); Yang und Yang (1997) zu entnehmen. Das allgemeine CSP lässt sich auf die folgende, deklarative Weise definieren (vgl. Kolbe 2000, S. 36):

Definition 5.2.1 (Constraint Satisfaction Problem, CSP)
Ein Constraint Satisfaction Problem wird durch ein Tripel (V, D, C) *beschrieben, wo-*

[1]**C**ylindrical **A**lgebraic **D**ecomposition

bei $V = \{v_1, \ldots, v_n\}$ *eine endliche Menge von Variablen mit assoziierten Wertebereichen* $D = \{D_1, \ldots, D_n\}$ *mit* $\{v_1 : D_1, \ldots, v_n : D_n\}$ *ist.* C *ist eine endliche Menge von Constraints* $C_j(V_j)$, $j \in \{1, \ldots, m\}$, *wobei jedes Constraint* $C_j(V_j)$ *eine Teilmenge* $V_j = \{v_{j_1}, \ldots, v_{j_k}\} \subseteq V$ *der Variablen zueinander in Relation setzt und deren gültige Wertekombinationen auf eine Teilmenge von* $D_{j_1} \times \cdots \times D_{j_k}$ *beschränkt.*

Zu beachten ist bei dieser allgemeinen Definition, dass die Wertebereiche der Variablen eines CSP nicht unbedingt auf Zahlenwerte beschränkt sein müssen. So können durchaus andere (symbolische) Domänen zum Einsatz kommen, die sich letztendlich allerdings durchaus wiederum als Zahlenwerte repräsentieren lassen.

Klassischerweise bezieht sich die Literatur bei der Definition von CSPs häufig auf eine strengere Form, in der die Domänen der Variablen aus diskreten, endlichen Mengen bestehen (vgl. Mackworth 1977a; Waltz 1975). Auch heute beschränken sich viele Lehrbücher und wissenschaftliche Veröffentlichungen bei der Definition von CSPs ausschließlich auf finite Domänen (vgl. Barták 2001; Kumar 1992; Tsang 1993; Van Hentenryck 1989). In der Vergangenheit wurden CSPs mit finiten Domänen in Anlehnung an Waltz (1975) auch als *Consistent Labeling Problem* benannt (vgl. Haralick und Shapiro 1979; Tsang 1993).[2] In neueren Publikationen wird basierend auf dem Terminus von Mackworth (1992) zwischen dem allgemeinen CSP und dem *Finite Constraint Satisfaction Problem* (FCSP) unterschieden (vgl. Mackworth und Freuder 1993; Meyer 1995; Tsang 1993), welches wie folgt definiert wird (vgl. Meyer 1995, S. 28):

Definition 5.2.2 (Finite Constraint Satisfaction Problem, FCSP)
Sei $P = (V, D, C)$ *ein CSP. Wenn die Domäne* D_i *jeder Variablen* $v_i \in V$ *diskret und endlich ist, wird* P *ein Finite Constraint Satisfaction Problem (FCSP) genannt.*

Weil bei einem FCSP die Wertebereiche der Variablen endlich sind, ist auch der Lösungsraum endlich. Die Anzahl der möglichen Lösungen ergibt sich wiederum aus dem kartesischen Produkt aller Wertebereiche $D_1 \times \cdots \times D_n$. Die Kardinalität dieser Menge, und entsprechend auch der Aufwand zur Berechnung dieser möglichen Lösungen, wächst exponentiell mit der Anzahl der Variablen (vgl. Kolbe 2000, S. 37). Das FCSP gehört zur Klasse der NP-vollständigen Probleme.[3] Der formale Beweis für die NP-Vollständigkeit wird von Haralick und Shapiro (1979, S. 177) durch die Reduzierung des FCSP auf das SAT-Problem geführt.[4]

Eine weitere wichtige Klasse sind CSPs, die nur aus unären und binären Constraints bestehen. Sie lassen sich als Graph darstellen, indem die Variablen die Knoten und die

[2] Die einzelnen Elemente der Wertebereiche einer Variable werden von Waltz (1975) als *label* (engl.: Marke, Markierung) bezeichnet.

[3] NP-vollständige Probleme gehören zu den schwierigsten Problemen in der Informatik. Ein Problem, das NP-vollständig ist, lässt sich nur von einem *nichtdeterministischen* Algorithmus in polynomialer Laufzeit lösen. Bisher konnte nicht gezeigt werden, dass ein NP-vollständiges Problem *deterministisch* in Polynomialzeit lösbar ist, womit ein offenes Problem der Informatik, nämlich ob $P = NP$ ist, gelöst wäre. Es besteht die starke Vermutung, dass $P \neq NP$ gilt, und somit kein NP-vollständiges Problem durch ein effizientes Verfahren gelöst werden kann (vgl. Claus und Schwill 2001, S. 448 ff.; Broy 1995, S. 122).

[4] SAT: *satisfiability problem*, Erfüllbarkeitsproblem der Aussagenlogik (vgl. Claus und Schwill 2001, S. 230).

binären Constraints die Kanten zwischen den Knoten repräsentieren. Unäre Constraints werden i. d. R. nicht explizit dargestellt, da sie den zugewiesenen Wertebereichen der Variablen entsprechen. Ein solcher Constraint-Graph wird entsprechend Constraint-Netz genannt. Ein binäres CSP wird wie folgt definiert (vgl. Tsang 1993, S. 12):

Definition 5.2.3 (binäres CSP, allgemeines CSP)
Ein binäres Constraint Satisfaction Problem, oder binäres CSP, ist ein CSP mit ausschließlich unären und binären Constraints. Ein CSP, das nicht auf unäre und binäre Constraints beschränkt ist, wird allgemeines CSP genannt.

Die große Mehrzahl der Veröffentlichungen beschränkt sich auf die Behandlung von binären Constraints und entsprechend binären CSPs, was darin begründet liegt, dass zum einen diese Voraussetzung die Entwicklung von effizienten Lösungsverfahren vereinfacht und zum anderen jedes höherwertige Constraint mit einer Stelligkeit größer als zwei auf relativ einfache Weise in ein binäres Constraint überführt werden kann.

Constraints mit einer Stelligkeit höher als zwei werden auch n-äre Constraints bzw. *non-binary* oder *high-order* Constraints genannt. Spezielle Verfahren zur direkten Behandlung von n-ären Constraints bzw. zur Umwandlung von n-ären in binäre Constraints, werden in Abschnitt 5.2.6 auf Seite 135 ff. vorgestellt.

5.2.1 Historisches Beispiel

Erstmalig wurde ein Problem als CSP im Rahmen einer Teilaufgabe der automatischen Bilderkennung von Waltz (1972)[5] bzw. Waltz (1975) behandelt. Die Aufgabe bestand für *David L. Waltz* darin, den dreidimensionalen Aufbau einer Szene (*Blocks World*) anhand einer Linienzeichnung zu bestimmen, von dem lediglich die zweidimensionale Kantenstruktur bekannt ist. Da die Linien der Szene als Objektkanten entsprechend „markiert" werden, wird dieses spezielle CSP auch *Consistent Labeling Problem* bzw. *Scene Labeling Problem* genannt. Um die einzelnen Objekte bestimmen zu können, müssen die Linien der Szene folgendermaßen klassifiziert werden:[6]

- **Konvexe Kanten** sind gegeben, wenn die beiden angrenzenden Ebenen von dem Betrachter weggerichtet sind. Konvexe Kanten werden mit einem „+" markiert.
- **Konkave Kanten** sind Kanten, deren angrenzenden Ebenen auf den Betrachter zustreben. Sie sind mit einem „–" markiert.
- **Verdeckende Kanten** sind konvexe Kanten, von denen nur eine angrenzende Ebene sichtbar ist, weil sie die andere Ebene verdeckt. Diese Kanten werden durch Pfeilsymbole („←", „→") markiert, wobei sich, bezogen auf die Pfeilrichtung, die für den Betrachter sichtbare Ebene jeweils rechts von der Kante befindet.

[5] Zit. nach Chmeiss und Jégou (1998, S. 124); Freuder (1978, S. 959); Güsgen (1989, S. 22); Güsgen (2000, S. 267); Lhomme (1993, S. 235); Mackworth (1977a, S. 105); Meyer (1995, S. 49); Van Hentenryck (1989, S. 45).

[6] Das von Waltz (1975) beschriebene System ist zudem in der Lage, eine Reihe weiterer Kanten zu klassifizieren, z. B. zur Interpretation von Schatten.

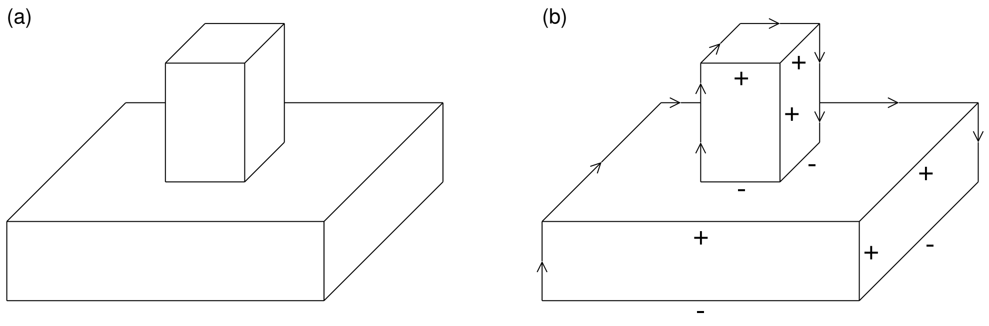


Abbildung 5.1: Beispiel und mögliche Lösung eines Labeling Problems

Verdeckende Kanten werden auch „Außenkanten", konvexe und konkave Kanten entsprechend „Innenkanten" genannt. In Abbildung 5.1 ist ein Beispiel für eine Linienzeichnung (a) sowie das Ergebnis der Klassifikation (b) zu sehen.

Zur Abbildung des Problems auf ein CSP wird jede Kante als eine Variable repräsentiert. Die Domäne jeder Variable besteht in der Ausgangssituation aus einer endlichen Menge der möglichen Klassen $\{+, -, \leftarrow, \rightarrow\}$. Die Menge der Constraints ergibt sich aus einer Reihe von allgemeinen, geometrischen Regeln. So lässt sich, bezogen auf die Eckpunkte, eine Aussage über die Markierungen der zusammentreffenden Kanten treffen: Die Constraints schränken die Anzahl der möglichen Markierungen der Szene auf zulässige Kombinationen ein, so dass sich eine korrekte, dreidimensionale Interpretation ergibt.

Da in der Praxis die Anwendung eines Suchalgorithmus aufgrund der Komplexität nicht adäquat ist, entwickelte Waltz eine Alternative, den „Waltz-Filteralgorithmus". Dieser Algorithmus nimmt eine *lokale* Auswertung der Constraints vor, indem inkompatible Belegungen aus den Domänen der Constraint-Variablen herausgefiltert werden. Die Einschränkungen der Wertebereiche werden in den benachbarten Constraints propagiert, ohne dabei eine Menge von allen möglichen Kombinationen der Markierungen zu erzeugen. Dadurch wird eine effiziente Verarbeitung des *Consistent Labeling Problems* gewährleistet. Eine übersichtliche Darstellung des Waltz-Filteralgorithmus findet sich z. B. in den Arbeiten von Meyer (1995, S. 15) und Güsgen (1989, S. 25 f.).[7]

5.2.2 Lösen eines CSP

Mit *Constraint Satisfaction* wird die Suche nach einer global konsistenten Wertebelegung für alle Constraint-Variablen eines CSP bezeichnet, d. h. es müssen alle Constraints erfüllt sein. Ein Constraint wird erfüllt, wenn die Wertezuweisungen zu den jeweils involvierten Variablen den Anforderungen der Constraint-Relation genügen (vgl. Güsgen 2000, S. 269):

Definition 5.2.4 (Constraint-Erfüllung)
Gegeben sei ein CSP mit den Constraints $C_1, \ldots, C_m$ auf den Variablen $v_1, \ldots, v_n$ mit den

[7]Das Verfahren erzeugt einen Konsistenzgrad, der dem der *Kantenkonsistenz* entspricht. Der Waltz-Filteralgorithmus selbst ist äquivalent zum `AC-2` Algorithmus (vgl. Abschnitt 5.2.3.3, S. 92).

Wertebereichen $D_1, \ldots, D_n$. *Ein Tupel* $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ *erfüllt ein Constraint* C_j, $j \in \{1, \ldots, m\}$, *falls die zu den Variablen von* C_j *gehörenden Werte aus* $(d_1, \ldots, d_n)$ *ein Element der Relation von* C_j *bilden.*

Es genügt allerdings nicht, wenn jedes Constraint für sich genommen gültige Wertekombinationen aufweist, d. h. konsistent ist. Dies ist aber häufig die Vorstufe, um zu einer globalen Lösung eines CSP zu gelangen, welche „aus einer eindeutigen Belegung der Variablen mit genau einem Wert aus dem jeweiligen Wertebereich besteht, so dass alle Constraints erfüllt sind“ (Kolbe 2000, S. 36). Die Lösung eines CSP lässt sich formal folgendermaßen definieren (vgl. Güsgen 2000, S. 269):

Definition 5.2.5 (Lösung eines CSP)
Gegeben sei ein CSP mit den Constraints $C_1, \ldots, C_m$ *auf den Variablen* $v_1, \ldots, v_n$ *mit den Wertebereichen* $D_1, \ldots, D_n$. *Eine Lösung des CSP ist ein Tupel* $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$, *das* $C_1, \ldots, C_m$ *erfüllt.*

Das Auflösen eines CSP stellt sich als Suche im Lösungsraum dar, welcher durch alle möglichen Lösungen aufgespannt wird. Jede Variable kann dabei Werte aus ihrem Wertebereich annehmen.[8] Je nach Fragestellung können unterschiedliche Ergebnisse erwünscht sein. So kann sich zum einen die Frage nach der Existenz einer Lösung stellen, zum anderen, wenn es mehrere Lösung gibt, die Frage nach deren Anzahl bzw. nach deren konkreten Ausprägungen. Weiterhin ist bei Optimierungsproblemen die Frage nach der besten Lösung von Interesse (vgl. Dechter 1999, S. 195). Da bei einem CSP die Frage nach der Lösbarkeit gleichbedeutend mit der Suche nach der ersten Lösung ist, reduziert sich die Aufgabenstellung hier auf die Suche nach einer Lösung bzw. nach allen möglichen Lösungen (vgl. Kolbe 2000, S. 37). Die Berechnung erfolgt durch spezielle Lösungsverfahren, die innerhalb von Constraint-Solvern implementiert werden. Durch Constraint-Solver wird die Trennung von Problemstellung und Lösungsverfahren realisiert, da sie generelle Mechanismen zum Auflösen von kombinatorischen Problemen, formalisiert als CSP, bieten. In ihnen können unterschiedliche Algorithmen Verwendung finden, die je nach Bedarf für bestimmte Problemstellungen optimiert sein können, und die ebenfalls nach Bedarf ein entsprechendes Antwortverhalten bieten. Die meisten Algorithmen zum Lösen eines CSP sind Varianten bzw. Kombinationen von Lösungsverfahren, die sich in zwei Klassen einteilen lassen: *Konsistenzverfahren* und *Suchverfahren*:

Konsistenzverfahren werden von vielen Constraint-Solvern zum effizienten Auflösen von CSPs eingesetzt, um eine Reduzierung des Lösungsraums vorzunehmen, bevor sie eine global gültige Lösung suchen. Sie eliminieren ungültige Werte aus den Wertebereichen der Constraint-Variablen und stellen dadurch einen Grad *lokaler Konsistenz* her. Constraint-Solver, die lokale Konsistenz herstellen, sind korrekt, d. h. es gehen keine Lösungen verloren. Gleichzeitig sind sie unvollständig, da sie für

[8] Diese simultane Zuweisung von Werten zu den entsprechenden Variablen wird von Tsang (1993, S. 6) explizit als *compound label* definiert. Nach dieser Auffassung kann ein Constraint auch extensional als Menge von *compound labels* gesehen werden.

gewöhnlich nicht alle ungültigen Werte aus den Wertebereichen herausfiltern. Dafür ist ihr Laufzeitverhalten i. d. R. auf polynomiale Laufzeitkomplexität beschränkt (vgl. Mackworth und Freuder 1985, S. 66; Marriott und Stuckey 1999, S. 89). Nur in seltenen Fällen liegt nach dem Herstellen lokaler Konsistenz eine eindeutige Lösung vor. Vielmehr sind die Wertebereiche der Constraint-Variablen auf die Werte beschränkt worden, die potentiell Teil einer Lösung sind. Für manche Anwendungen, die z. B. außer Constraints noch weitere Inferenzmechanismen nutzen, kann dies bereits ausreichend sein. Für eine Konfigurierungsaufgabe können auf diesem Weg bereits ein Großteil inkonsistenter Variablenbelegungen herausgefiltert werden. Häufig werden konsistenzbasierte Verfahren mit einem darauf folgenden Suchalgorithmus kombiniert, bzw. werden bei jedem Schritt eines Suchalgorithmus dynamisch Konsistenztests zur Verkleinerung der zu durchsuchenden Wertebereiche ausgeführt, um zu global gültigen Lösungen zu gelangen (vgl. Güsgen 2000, S. 275 ff.).

Suchverfahren lassen sich in *systematische* und *stochastische* Verfahren einteilen. Systematische Suchverfahren basieren i. d. R. auf einem *Backtracking*-Ansatz. Beim einfachen, chronologischen Backtracking werden in einer *Brute-Force*-Suche alle möglichen Wertekombinationen für eine globale Lösung eines FCSP durch Aufzählen ermittelt.[9] Bei inkonsistenten Teilbelegungen erfolgt ein Zurücksetzen zum letzten konsistenten Zustand (engl. *backtracking*). Dieses systematische Vorgehen nimmt im ungünstigsten Fall exponentiellen Aufwand in Kauf, da der Aufwand für das Lösen von FCSP NP-vollständig ist. Für Backtracking und seine Varianten spricht, dass die Algorithmen vollständig sind, d. h. sie können immer entscheiden, ob das FCSP Lösungen besitzt, und welche das sind (vgl. Dechter und Frost 2002; Kumar 1992; Marriott und Stuckey 1999).

Stochastische Suchverfahren werden oftmals auch lokale Suchverfahren (engl. *local search*), (heuristische) Reparaturverfahren oder GSAT-Verfahren (*Greedy SATisfiability*) genannt. Im Gegensatz zu systematischen Verfahren, folgen stochastische Suchverfahren keinem „Pfad“ zur schrittweisen Belegung der Constraint-Variablen mit Werten durch den Suchraum, sondern bewegen sich aufgrund von lokalen Kriterien und Zufallswerten. Die Suche beginnt mit einer zufälligen Wertezuweisung an alle Variablen. Schritt für Schritt wird diese Wertezuweisung durch Auswertung der vorliegenden Relationen „verbessert“, bis keine bzw. möglichst nur noch wenige Konflikte vorliegen. Stochastische Verfahren finden häufig auch in sehr großen Suchräumen relativ schnell eine Lösung. Zudem liegt zu jedem Zeitpunkt eine, wenn auch suboptimale, Lösung vor, da alle Variablen mit Beginn der Suche mit einem Wert belegt sind.[10] Allerdings besteht die Gefahr, dass die Algorithmen bereits bei einem lokalen Minima terminieren und keine global gültige Lösung finden.[11] Stochastische Verfahren sind heuristische, nichtdeterministische und unvollständige Suchverfahren,

[9] Tiefensuche in einem Suchbaum.

[10] Die Algorithmen, die mittels einer stochastischen Suche arbeiten, werden deshalb auch *anytime*-Algorithmen genannt.

[11] Das heißt es sind nicht alle Constraints erfüllt. In diesem Fall, muss der Suchalgorithmus entweder neu beginnen, oder mit Hilfe einer Heuristik versuchen, das lokale Minima zu verlassen.

d. h. sie können im Gegensatz zu systematischen Verfahren nicht garantieren, dass eine oder gar alle Lösungen eines Problems gefunden werden. Beispiele für bekannte Ansätze sind *Hill Climbing*, *Min-Conflicts*, *Simulated Annealing*, *Tabu-Suche* und *Genetische Algorithmen* (vgl. Barták 1999a, b; Minton et al. 1992; Russel und Norvig 2002; Tsang 1993).

Für bestimmte Anwendungen, für die frühzeitig eine teilweise korrekte Lösung nützlicher ist, als eine vollständige Lösung, deren Berechnung um ein vielfaches länger dauern würde, können unvollständige Suchverfahren sinnvoll sein. Sie stehen allerdings nicht im Fokus dieser Arbeit. Für Konfigurierungsaufgaben kann eine Reduktion des Lösungsraums durch Konsistenztechniken zum Ausschließen einer Vielzahl von fehlerhaften Konfigurationen sehr sinnvoll sein, wenn es, wie in EngCon, übergeordnete Inferenzmechanismen zur weitergehenden Konfigurierung gibt. Ebenfalls wünschenswert ist die Berechnung einer oder aller globaler Lösungen, bzw. einer möglicherweise sogar optimalen Lösung, als Ergebnis eines vollständigen Suchverfahrens. Fatal für den Konfigurierungsvorgang könnte es sich allerdings auswirken, wenn der Constraint-Solver potentiell falsche Lösungen als Ergebnis einer unvollständigen Suche liefern würde. Diese suboptimalen Lösungen, in denen nicht alle Constraints erfüllt sind, würden, wenn dem nicht durch gesonderte Mechanismen Rechnung getragen wird,[12] unweigerlich in Konfigurierungskonflikten münden.

Daneben gibt es noch weitere spezielle, strukturbasierte Verfahren zum Auflösen von FCSPs. Strukturbasierte Verfahren nutzen neben Such- und Konsistenzverfahren besondere Eigenschaften der Topologie des Constraint-Graphen, um effizient zu einer Lösung zu gelangen (vgl. Dechter 1999, S. 196; Van Hentenryck und Saraswat 1996, S. 708). Bekannte Verfahren sind das *Tree-Clustering* (vgl. Dechter und Pearl 1989) und die *Cycle-Cutset*-Methode, auch *Cutset Conditioning* (vgl. Dechter 1990a, S. 292 ff.) genannt. In Abhängigkeit von der Problemstellung können mit diesen Verfahren gute Ergebnisse erzielt werden. Sie sind allerdings ebenso wie stochastische Suchverfahren stark abhängig von der Problemstellung, da die Effektivität dieser Verfahren von der Struktur des jeweiligen Constraint-Graphen und den zum Einsatz kommenden Heuristiken abhängt. Sie sind daher nicht so „stabil“ bzw. „robust“ wie systematische Suchverfahren, die durch Konsistenztechniken unterstützt werden, insbesondere wenn alle Lösungen eines Problems gesucht werden.

Auf eine genauere Betrachtung von stochastischen und strukturbasierten Verfahren wird daher in dieser Arbeit verzichtet. Konsistenzverfahren und systematische Suchverfahren zur Auflösung von FCSP, bzw. der Kombination beider, wird in den folgenden beiden Abschnitten der Vorzug gegeben. Im nachfolgenden Abschnitt 5.2.3 werden eine Reihe von lokalen Konsistenzgraden und Algorithmen zu deren Herstellung vorgestellt. In Abschnitt 5.2.4 auf Seite 111 ff. hingegen werden systematische Suchverfahren zum Auffinden von globalen Lösungen beschrieben.

[12] Beispielsweise durch Constraint-Hierarchien oder partielles *Constraint Satisfaction* (vgl. Abschnitt 4.4.3, S. 59).

5.2.3 Konsistenzverfahren

Die Belegung zweier Variablen ist konsistent, wenn alle Bedingungen (Constraints) an diesen beiden Variablen erfüllt sind. Analog sind drei Variablen konsistent, wenn sie mit Werten aus ihren jeweiligen Wertebereichen belegt sind, so dass alle betroffenen Constraints erfüllt sind etc. Konsistenzbasierte Constraint-Solver nehmen eine Domänenreduktion vor, indem sie mittels entsprechender Algorithmen Konsistenztests durchführen und ungültige Werte aus den Domänen der Constraint-Variablen entfernen. Das CSP wird dadurch in ein äquivalentes, einfacheres CSP umgeformt, in dem sich in nachfolgenden Schritten leichter eine Lösung finden lässt. Je nach gefordertem Konsistenzgrad sind diese Konsistenztests entsprechend aufwendig.

Die Anwendung von Konsistenztechniken ist eine *a priori* Reduktion des Suchraums, mit dem Ziel Inkonsistenzen durch ungültige Variablenbelegungen weitestgehend auszuschließen. Wenn durch diesen Vorgang die Domäne einer Constraint-Variable keine Elemente mehr enthält, ist frühzeitig erkennbar, dass sich das CSP offensichtlich nicht lösen lässt (engl. *domain wipe out*). Den Constraints fällt dabei eine aktive Rolle zu. Sie sind aktive Komponenten im Constraint-Netz, die aufgrund ihrer Eigenschaften die Wertebereiche der ihnen zugeordneten Variablen beschränken. Dadurch, dass die Constraints über ihre Variablen im Constraint-Netz verbunden sind, können Einschränkungen der Wertebereiche durch das Netz propagiert werden.

5.2.3.1 Constraint-Propagation

Um gültige Werte durch Abhängigkeiten zu erzeugen bzw. um die Konsistenz bzgl. der Constraint-Relationen zu überprüfen, werden Wertebereichsänderungen der Constraint-Variablen durch das Constraint-Netz propagiert. *Constraint-Propagation* bedeutet, dass sich Änderungen bzgl. des Wertebereichs einer Constraint-Variable an einer bestimmten Stelle im Constraint-Netz durch die Verknüpfungen der Constraints, und den in ihnen definierten Abhängigkeiten, auf andere involvierte Variablen bzw. ihrer Wertebereiche, auswirken:[13]

Definition 5.2.6 (Constraint-Propagation)
Constraint-Propagation bezeichnet das sukzessive Einschränken der Wertebereiche der Variablen des Constraint-Netzes durch Auswerten der Constraints. Lokale Änderungen der Wertebereiche breiten sich durch wiederkehrende Propagation durch das gesamte Constraint-Netz aus und schränken so den Lösungsraum immer weiter ein.

Die Beschränkung des Wertebereichs einer Variable hat Auswirkungen auf die Wertebereiche der übrigen Variablen, die mit dieser Variable im Constraint-Netz über ein Constraint in Verbindung stehen, d. h. wenn sie in demselben Constraint vorkommen.

[13] Die hier verwendete Beschreibung von Constraint-Propagation entspricht der klassischen Verwendung des Begriffs (vgl. z. B. Freuder 1978; Mackworth 1977a). In neueren Arbeiten, wird mit Constraint-Propagation z. T. auch die Kombination von Konsistenz- und Suchverfahren benannt (vgl. Barták 1999a, S. 559; Barták 1999b, S. 11).

Deren eingeschränkten Wertebereiche schränken wiederum die Wertebereiche anderer Variablen ein usw. Die Änderungen breiten sich durch das Constraint-Netz aus: sie werden *propagiert*. Es verringert sich dadurch insgesamt die Anzahl der möglichen Belegungen.

Das Verfahren wurde erstmals von *David L. Waltz* zur automatischen Erkennung von Linienzeichnungen eingesetzt. Waltz (1975) verwendete die Propagationsmethode zur Interpretation von Linienzeichnungen dreidimensionaler Objekte (vgl. Abschnitt 5.2.1, S. 85). Von *Alan K. Mackworth* wurde die Idee des Propagierens aufgegriffen und zum allgemeinen Konzept der Konsistenztechniken weiterentwickelt (vgl. Mackworth 1977a).[14] Im Wesentlichen werden eine Reihe von Konsistenzgraden unterschieden, die in Anlehnung an die Terminologie der Graphentheorie *Knotenkonsistenz*, *Kantenkonsistenz*, *Pfadkonsistenz* und *k-Konsistenz* genannt werden. In den folgenden Abschnitten werden diese Konsistenzgrade jeweils erläutert. Zu beachten ist, dass zur Herstellung einer bestimmten Konsistenz grundsätzlich unterschiedliche Algorithmen verwendet werden können.

5.2.3.2 Knotenkonsistenz

Der einfachste Konsistenzgrad ist die *Knotenkonsistenz*. Dieser Konsistenzgrad bezieht sich ausschließlich auf unäre Constraints, d. h. im Regelfall auf einfache Wertezuweisungen. Knotenkonsistenz ist folgendermaßen definiert (vgl. Güsgen 2000, S. 270):

Definition 5.2.7 (Knotenkonsistenz/node consistency, NC)
Gegeben sei ein beliebiges CSP auf den Variablen $v_1, \ldots, v_n$ mit den Wertebereichen $D_1, \ldots, D_n$. Sei $C_1^u, \ldots, C_n^u$ die Menge der unären Constraints im Constraint-Netz, wobei C_i^u, $i \in \{1, \ldots, n\}$, die Relation R_i^u besitzt und die Variable v_i einschränkt. Das Constraint-Netz ist knotenkonsistent, falls für alle $i \in \{1, \ldots, n\}$ gilt: $D_i \subseteq R_i^u$.

Ein Constraint-Netz ist dementsprechend knotenkonsistent, wenn alle unären Constraints erfüllt sind, d. h. wenn sich alle vorhandenen unären Constraints trivialerweise in den Wertebereichen der Variablen bereits widerspiegeln. In Abbildung 5.2 auf der nächsten Seite ist der Algorithmus `NC-1` zur Herstellung von Knotenkonsistenz von Mackworth (1977a) dargestellt. Der Algorithmus erhält als Eingabe die Menge der unären Constraints $C_1^u, \ldots, C_n^u$. In der Prozedur `NC` werden der Reihe nach alle Domänen D_i auf den Wert d_i beschränkt, der mit dem jeweiligen Constraint C_i^u konsistent ist. Das Laufzeitverhalten des Algorithmus ist linear mit der Anzahl der Variablen und der Anzahl der möglichen Werte in den Wertebereichen, da der Algorithmus jeden Knoten im Constraint-Netz mit den möglichen Werten lediglich einmal inspiziert.

Knotenkonsistenz beschränkt in einem Constraint-Netz die Wertebereiche der Variablen auf die durch unäre Constraints erlaubten Werte und schließt dadurch u. U. bereits einen großen Teil Werte aus, die nicht zur Lösung des CSP beitragen können, da sie einzelne Constraints verletzen.

[14] In englischsprachigen Veröffentlichungen auch *discrete relaxation* und *constraint relaxation* genannt (vgl. Dechter und Pearl 1989, S. 354; Freuder 1978, S. 963; Haralick und Shapiro 1979, S. 178; Mohr und Masini 1988, S. 651).

```
procedure NC(v_i):
D_i ← D_i ∩ {d_i | C_i^u(d_i)}

begin
  for i ← 1 until n do NC(v_i)
end
```

Abbildung 5.2: Algorithmus `NC-1` zur Herstellung von Knotenkonsistenz (vgl. Mackworth 1977a, S. 103)

5.2.3.3 Kantenkonsistenz

Ein höherer, lokaler Konsistenzgrad ist die *Kantenkonsistenz*. Im Gegensatz zur Knotenkonsistenz bezieht sich die Kantenkonsistenz nicht auf die Wertebereiche einzelner Knoten im Constraint-Netz, sondern auf die Domänen der Knoten, die durch eine Kante verbunden sind, d. h. über ein Constraint zueinander in Relation stehen. Für gewöhnlich bezieht sich Kantenkonsistenz aus Gründen der Vereinfachung auf binäre Constraint-Netze.

Kantenkonsistenz stellt sicher, dass die Wertebereiche der Variablen auf die Werte beschränkt werden, die kompatibel zu unmittelbar benachbarten Variablen sind. Zu jedem möglichen Wert einer Variable muss ein passender Wert in den Wertebereichen der Variablen vorhanden sein, die zu dieser Variable über ein Constraint in Relation stehen.[15] Formal lässt sich dies wie folgt ausdrücken (vgl. Güsgen 2000, S. 270):

Definition 5.2.8 (Kantenkonsistenz/arc consistency, AC)
Gegeben sei ein binäres, knotenkonsistentes CSP auf den Variablen $v_1, \ldots, v_n$ mit den Wertebereichen $D_1, \ldots, D_n$. Sei $C^b_{1,1}, \ldots, C^b_{n,n}$ die Menge der binären Constraints im Constraint-Netz, wobei $C^b_{i,j}$, $i, j \in \{1, \ldots, n\}$ und $i \neq j$, die Relation $R^b_{i,j}$ besitzt und die Variablen v_i und v_j einschränkt. Das Constraint-Netz ist kantenkonsistent, falls für alle $i, j \in \{1, \ldots, n\}$ mit $i \neq j$ gilt: $\forall d_i \in D_i\ \exists d_j \in D_j\ [(d_i, d_j) \in R^b_{i,j}]$.

Ein Constraint-Netz ist kantenkonsistent, wenn alle Constraints kantenkonsistent sind. Erreicht wird dies, indem Änderungen der Wertebereiche der Constraint-Variablen mittels entsprechender Algorithmen durch das Netz propagiert werden. Alleine durch Herstellung der Kantenkonsistenz kann im Regelfall allerdings keine eindeutige Lösung für ein CSP bestimmt werden, bzw. lässt sich nicht eindeutig feststellen, dass es keine Lösung gibt.

Beispiel 5.2.1 *Für das CSP mit den Variablen*

$$v_1 : \{1, 2\}, \quad v_2 : \{1, 2\}, \quad v_3 : \{1, 2\}$$

und den Constraints

$$C_{1,2} : v_1 \neq v_2, \quad C_{2,3} : v_2 \neq v_3, \quad C_{1,3} : v_1 \neq v_3$$

[15] Für den allgemeinen Fall der ungerichteten Kantenkonsistenz muss dies auch andersherum gelten.

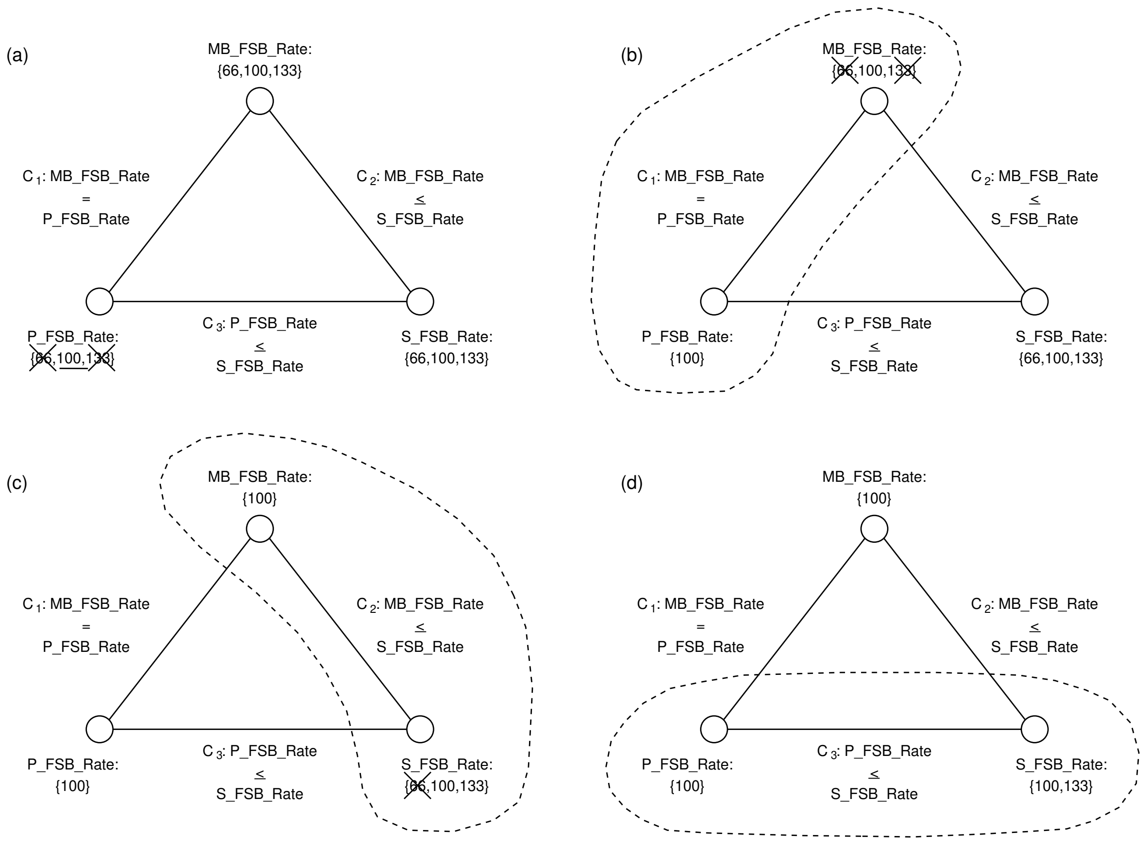


Abbildung 5.3: Propagation zur Herstellung von Kantenkonsistenz

gibt es keine eindeutige Lösung, obwohl es Kantenkonsistent ist. Für jede „Kante“ bzw. jedes Constraint gibt es für sich genommen mindestens eine lokal konsistente Wertekombination, z. B. $v_1 = 1$ *und* $v_2 = 2$ *für* $C_{1,2}$*. Entsprechend müsste für das Constraint* $C_{2,3}$ *die Variable* $v_3 = 1$ *sein, was aber inkonsistent mit* $C_{1,3}$ *ist, da* v_1 *ebenfalls 1 ist.*

In Abbildung 5.3 ist die zur Erzeugung von Kantenkonsistenz notwendige Propagation anhand des Constraint-Beispiels aus Abschnitt 3.6.3 auf Seite 39 f. dargestellt. In (a) wird während eines Konfigurierungsvorgangs die Taktfrequenz des Prozessors `P_FSB_Rate` mit 100 festgelegt. Die Auswertung der Kante bzw. des Constraints C_1 bewirkt, dass der Wertebereich von `MB_FSB_Rate` ebenfalls auf den Wert 100 eingeschränkt wird (b). Die Auswertung des Constraint C_2 in (c) schränkt wiederum den Wertebereich von `S_FSB_Rate` ein. In (d) ist zu sehen, wie durch Auswertung von C_3 sichergestellt wird, dass die Wertebereiche von `P_FSB_Rate` und `S_FSB_Rate` konsistent mit C_3 sind. Da dies der Fall ist, sind keine weiteren Wertebereichseinschränkungen erforderlich. Das Ergebnis der Propagation ist ein kantenkonsistentes Constraint-Netz, welches noch keine eindeutige, allerdings zwei potentielle Lösungen bietet.

Im Folgenden werden einige einfache Algorithmen von Mackworth (1977a) vorgestellt, mit deren Hilfe Kantenkonsistenz erzeugt werden kann. Der in Abbildung 5.4 darge-

procedure REVISE$((v_i, v_j))$:
begin
  DELETE $\leftarrow$ **false**
  for each $d_i \in D_i$ **do**
    if there is no $d_j \in D_j$ such that $C^b_{i,j}(d_i, d_j)$ **then**
      begin
        delete d_i from D_i;
        DELETE $\leftarrow$ **true**
      end;
  return DELETE
end

Abbildung 5.4: Algorithmus `REVISE` zur Entfernung inkompatibler Werte (vgl. Mackworth 1977a, S. 104)

begin
  for $i \leftarrow 1$ **until** n **do** NC(v_i);
  $Q \leftarrow \{(v_i, v_j) \mid (v_i, v_j) \in arcs(G), v_i \neq v_j\}$
  repeat
    begin
      CHANGE $\leftarrow$ **false**
      for each $(v_i, v_j) \in Q$ **do** CHANGE $\leftarrow$ (REVISE$((v_i, v_j))$ **or** CHANGE)
    end
  unil $\neg$ CHANGE
end

Abbildung 5.5: Der Kantenkonsistenz-Algorithmus `AC-1` (vgl. Mackworth 1977a, S. 104)

stellte Algorithmus `REVISE` kann dazu genutzt werden, in knotenkonsistenten, binären Constraint-Netzen zwischen den übergebenen Variablenpaaren $(v_i, v_j) \in D_i \times D_j$ Kantenkonsistenz herzustellen. Erreicht wird dies, indem der Algorithmus sämtliche möglichen Werte $d_i \in D_i$ der ersten Variable daraufhin untersucht, ob es für jeden Wert einen kompatiblen Wert in der Domäne D_j der zweiten Variable gibt. Wenn dies nicht der Fall ist, kann der Wert d_i aus der Domäne D_i gelöscht werden.[16] Der Algorithmus `REVISE` muss auf alle Kanten bzw. binären Constraints angewendet werden. Da er nicht symmetrisch ist, müssen neben den Variablenpaaren (v_i, v_j) ebenfalls die entsprechenden Paare (v_j, v_i) geprüft werden.

Nachdem der Algorithmus auf ein Paar (v_i, v_j) angewendet wurde, ist dieses Variablenpaar zwar kantenkonsistent, bleibt dies aber u. U. nicht. Wenn die Variable v_j ebenfalls zu anderen Variablen durch binäre Constraints in Relation steht, löscht der Algorithmus voraussichtlich aus der Domäne von v_j ebenfalls Werte, so dass anschließend wiederum der Wertebereich von v_i überprüft werden muss, um Kantenkonsistenz zu gewährleisten.

[16] Dies reduziert nicht die Anzahl der möglichen Lösungen des CSP, da der gelöschte Wert d_i auf keinen Fall zu einer konsistenten Lösung beitragen kann.

begin
  for $i \leftarrow 1$ **until** n **do** NC(v_i);
  $Q \leftarrow \{(v_i, v_j) \mid (v_i, v_j) \in arcs(G), v_i \neq v_j\}$
  while Q not empty **do**
    begin
      select and delete any arc (v_k, v_m) from Q;
      if REVISE((v_k, v_m)) **then** $Q \leftarrow Q \cup \{(v_i, v_k) \mid (v_i, v_k) \in arcs(G), i \neq k, i \neq m\}$
    end
end

Abbildung 5.6: Der Kantenkonsistenz-Algorithmus `AC-3` (vgl. Mackworth 1977a, S. 106)

Ein einfacher Durchlauf von `REVISE` ist demnach nicht ausreichend. Stattdessen muss der Algorithmus solange aufgerufen werden, bis keine Änderungen an den Domänen mehr vorgenommen wurden. Erst danach ist die Kantenkonsistenz des Constraint-Netzes sichergestellt. Ein Algorithmus, der diese Funktion übernimmt, ist der in Abbildung 5.5 auf der vorherigen Seite dargestellte, einfachste Kantenkonsistenz-Algorithmus `AC-1`. Der Algorithmus `AC-1` arbeitet allerdings nicht sehr effizient, da alle Variablenpaare erneut untersucht werden müssen, solange bei einem Durchlauf auch nur eine einzige Änderung an der Domäne einer Variable vorgenommen wurde.

Eine Verbesserung ist der Algorithmus `AC-2` (vgl. Mackworth 1977a, S. 106), der dem von Waltz (1975) beschriebenen Filteralgorithmus entspricht. `AC-2` erreicht Kantenkonsistenz mit nur einem einzigen Durchgang der äußeren Schleife durch die Knoten, benötigt dafür allerdings neben Q eine zweite dynamische Liste Q'. Aufgrund der Eigenschaften des Algorithmus `AC-2` kann es bei Zyklen > 2 im Constraint-Netz dazu kommen, dass eine Kante in beiden Listen Q und Q' gleichzeitig auf Bearbeitung wartet (vgl. Mackworth 1977a, S. 107). Dies unterschiedet den Algorithmus von dessen Nachfolger `AC-3`. Der `AC-3`-Algorithmus (siehe Abbildung 5.6) ist eine Vereinfachung bzw. Verallgemeinerung von `AC-2`. Er ist nicht mehr darauf ausgelegt, Kantenkonsistenz mit einem einzigen Durchlauf herzustellen. Es wird lediglich eine einzige dynamische Liste Q verwaltet, die wie in `AC-1` alle Kanten des Constraint-Netzes enthält.[17] Bei Reduzierung des Wertebereichs von v_k durch `REVISE` müssen alle davon betroffenen Kanten (v_i, v_k) an die Liste Q angefügt werden.[18] Dies wird solange wiederholt, bis keine Reduzierung der Wertebereiche mehr erfolgt und die Liste Q vollständig abgearbeitet wurde.

Auch der `AC-3`-Algorithmus ist noch keineswegs optimal. So werden auch durch diesen Algorithmus bei weiteren Durchläufen mit `REVISE` noch häufig Wertepaare geprüft, von denen bereits bekannt ist, dass sie kompatibel miteinander sind. Eine Reihe von Algorithmen befasst sich damit, diese Metainformationen zu verwalten und durch weniger Konsistenztests effizientere Laufzeiten zu erreichen (siehe Tabelle 5.1, S. 106). Der `AC-4`-Algorithmus

[17] Aufgrund der fehlenden Symmetrieeigenschaften von `REVISE` sowohl die „Hin-“ (v_i, v_j) als auch die „Rückrichtungen“ (v_j, v_i).

[18] Wobei auf die Rückrichtung (v_m, v_k) der ursprünglichen Kante verzichtet werden kann. Die Kante (v_m, v_k) kann nicht inkonsistent geworden sein, denn durch `REVISE` wurden aus D_m nur Werte gelöscht, für die es keine korrespondierenden Werte in D_k gibt.

von *Roger Mohr* und *Thomas C. Henderson* nutzt intern eine separate Datenstruktur zur extensionalen Constraint-Repräsentation (vgl. Mohr und Henderson 1986). Dadurch wird es möglich, `REVISE` nur für Wertepaare auszuführen, die von einer vorherigen Wertebereichseinschränkung betroffen sind: Ein Wert in einer Domäne gilt als *supported*, wenn für diesen Wert konsistente Wertebelegungen für alle übrigen Variablen existieren. Wenn in einer Domäne ein Wert gelöscht wird, müssen nicht mehr alle Werte in den Domänen der mit dieser Variablen über ein Constraint verbundenen Variablen getestet werden, sondern nur noch die Werte, die auf den gelöschten Wert über einen *support*-Eintrag angewiesen waren.

Während also `AC-3` bereits nur noch Wertepaare von Kanten prüft, die von Änderungen betroffen sind, reduziert `AC-4` nochmals die zu überprüfenden Wertepaare auf diejenigen, die direkt von einer Änderung betroffen sind. Dies resultiert in einer optimalen *worst-case* Zeitkomplexität, da nicht mehr Konsistenztests als notwendig vorgenommen werden. Allerdings wird diese Verbesserung mit stark erhöhter Platzkomplexität und einer im Durchschnitt dicht am *worst-case* befindlichen Zeitkomplexität erkauft, so dass der `AC-3`-Algorithmus in der Praxis häufig performanter als `AC-4` arbeitet, trotzdem er das schlechtere *worst-case* Laufzeitverhalten aufweist (vgl. Bessière 1994a, S. 180; Bessière und Cordier 1993, S. 108; Wallace 1993, S. 239 ff.).[19] Insbesondere in Anwendungen, in denen die Domänen der Constraint-Variablen sehr viele Werte aufweisen, ist aufgrund der erhöhten Platzkomplexität von `AC-4` der `AC-3`-Algorithmus vorzuziehen.

`AC-3` und `AC-4` gehören zu bisher den am weitesten verbreiteten Algorithmen zur Herstellung von Kantenkonsistenz. Es gibt noch eine Reihe weiterer Algorithmen, die allerdings in der Praxis bisher keine so große Verbreitung gefunden haben. So ist der `AC-5`-Algorithmus von *Pascal Van Hentenryck* et al. ein genereller Algorithmus, der sich auf andere Kantenkonsistenz-Algorithmen reduzieren lässt (vgl. Van Hentenryck et al. 1992). `AC-3` und `AC-4` können als Spezialisierungen von `AC-5` gesehen werden. `AC-5` besitzt eine besondere Bedeutung für CLP-Sprachen, da sich spezielle, hier vorkommende Klassen von Constraints mittels weiterer Spezialisierungen von `AC-5` auf besonders effiziente Weise verarbeiten lassen. `AC-5*` ist eine Verbesserung von `AC-5` in der Hinsicht, dass sich Spezialisierungen nicht nur bzgl. der Zeitkomplexität sondern auch bezogen auf die Platzkomplexität optimieren lassen (vgl. Liu 1996).

Eine Verbesserung von `AC-4` ist der `AC-6`-Algorithmus von *Christian Bessière* und *Marie-Odile Cordier*. `AC-6` besitzt bei verringerter Platzkomplexität dasselbe *worst-case* Laufzeitverhalten wie `AC-4` und arbeitet in der Praxis schneller als `AC-3` und `AC-4`. Der `AC-6`-Algorithmus berechnet für jeden Wert einer Variable zunächst lediglich eine konsistente Wertebelegung pro Constraint. Weitere konsistente Belegungen (bzw. *supports*) werden erst bei Bedarf erzeugt, d. h. wenn eine vorherige Belegung inkonsistent geworden ist. Die Technik von `AC-6` wird deshalb auch *lazy* oder *minimal support* genannt (vgl. Bessière 1994a; Bessière und Cordier 1993). Weitere Verbesserungen bringt die Berücksichtigung von zusätzlichem Wissen über Constraints, wie die Auswertung von Symmetrieeigenschaften, welches in die Erweiterungen `AC-6+` (vgl. Bessière 1994b) bzw. `AC-6++` (vgl. Bessière

[19] Begründet liegt dies u. a. darin, dass der `AC-4`-Algorithmus aufgrund von *support*-Einträgen selektiv ungültige Werte entfernt, während der `AC-3`-Algorithmus bei jeder Überprüfung eines Constraints sämtliche Werte prüft und so inkonsistente Werte ggf. früher erkennt und entfernt (vgl. Wallace 1993, S. 242).

und Régin 1995) eingeflossen ist. Die Bemühungen weitere Metainformationen zu berücksichtigen mündeten durch die Arbeiten von von *Christian Bessière*, *Eugene C. Freuder* und *Jean-Charles Régin* in dem Algorithmus `AC-7`, der bei gleicher Zeit- und Platzkomplexität wie `AC-6` durch geschicktes Ausnutzen der Bidirektionalität von Constraints[20] weitere Werteüberprüfungen vermeidet (vgl. Bessière et al. 1995, 1999a).[21]

Weil die Datenstrukturen von `AC-4`, `AC-6` und `AC-7` zur Vermeidung von doppelten Konsistenztests sehr komplex sind und der Verwaltungsaufwand die Effizienz der Algorithmen in der Praxis vermindert, wurde versucht den Algorithmus `AC-3` ohne zusätzliche Datenstrukturen zu verbessern. `AC-8` (vgl. Chmeiss und Jégou 1996b, 1998), $\texttt{AC-3}_d$ (vgl. van Dongen 2002a, b) und `AC-2000` (vgl. Bessière und Régin 2001a, b) sind überarbeitete und leicht verbesserte Versionen von `AC-3`, die wie der originale Algorithmus ohne aufwendige Datenstrukturen auskommen. Bei gleicher *worst-case* Komplexität wie `AC-3` vermindern sie in der Praxis die Anzahl der benötigten Konsistenztests.[22] `AC-3.1` (vgl. Zhang und Yap 2001) und `AC-2001` (vgl. Bessière und Régin 2001a, b) sind Verbesserungen von `AC-3`, die durch ähnliche Techniken mit sehr einfachen zusätzlichen Datenstrukturen wesentliche Verbesserungen erreichen. `AC-3.1` und `AC-2001` sind optimale Algorithmen, d. h. sie besitzen eine optimale *worst-case* Zeitkomplexität, bei gleicher Platzkomplexität wie `AC-6` und `AC-7`.

Die Komplexität von Kantenkonsistenz-Algorithmen ist für den *worst-case* für binäre Constraints aus theoretischen Gründen auf quadratische Größenordnungen begrenzt. Auch zukünftige Algorithmen können daran nichts ändern. Da mit zunehmendem Grad lokaler Konsistenz, die Komplexität exponentiell anwächst, wird häufig vermieden, einen Konsistenzgrad größer als den der Kantenkonsistenz herzustellen. Ein weiterer Konsistenzgrad allerdings, der für bestimmte Anwendungen eine wichtige Rolle spielt, ist die *Pfadkonsistenz*.

5.2.3.4 Pfadkonsistenz

Da Kantenkonsistenz nicht alle inkonsistenten Werte aus den Domänen der Constraint-Variablen ausschließt, und zudem nicht sichergestellt ist, dass ein kantenkonsistentes Constraint-Netz eine Lösung besitzt, wurden höhere Konsistenzgrade und Verfahren zu deren Herstellung entwickelt. Die natürliche Erweiterung der Kantenkonsistenz ist die *Pfadkonsistenz*. Zur Herstellung der Pfadkonsistenz werden *Pfade* im Constraint-Graphen betrachtet, wobei ein Pfad eine Sequenz von Constraint-Variablen bzw. Knoten im Con-

[20] Wenn ein Wert von v_i einen Wert von v_j „unterstützt", dann wird ebenso v_j von v_i „unterstützt": Sei $C_{i,j}$, $C_{j,i}$ und $d_i \in D_i$, $d_j \in D_j$, dann ist $C_{i,j}(d_i, d_j)$ gdw. $C_{j,i}(d_j, d_i)$.

[21] Tatsächlich sind die beiden unabhängig voneinander entwickelten Algorithmen `AC-6++` von Bessière und Régin (1995) und `AC-7` von Freuder (1995) identisch und wurden erstmalig zusammen in demselben Workshop auf der ECAI'94 vorgestellt (vgl. Prosser 1995b, S. 2).

[22] Wobei für $\texttt{AC-3}_d$, trotzdem dies kein optimaler Algorithmus ist, in Versuchen wesentliche Verbesserungen aufgezeigt wurden. $\texttt{AC-3}_d$ nutzt ebenfalls wie `AC-7` die Bidirektionalität von Constraints zur Vermeidung von unnötigen Werteüberprüfungen. Dabei arbeitet $\texttt{AC-3}_d$ weiterhin auf derselben Datenrepräsentation wie der `AC-3`-Algorithmus.

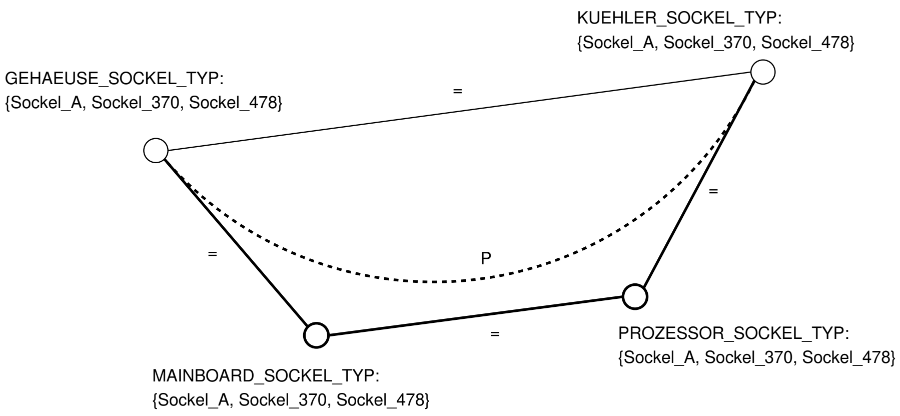


Abbildung 5.7: Pfade im Constraint-Netz

straint-Netz ist.[23] Dabei werden wie bei der Kantenkonsistenz Wertebelegungen eliminiert, die im Widerspruch zu den durch die Constraints formulierten Forderungen stehen.

Beispiel 5.2.2 *Bezogen auf das PC-Beispiel aus Kapitel 3 sind in Abbildung 5.7 die (symbolischen) Abhängigkeiten des Sockel-Typs der CPU auf andere Komponenten dargestellt. Neben dem Constraint zwischen* `GEHAEUSE_SOCKEL_TYP` *und* `KUEHLER_SOCKEL_TYP` *besteht eine indirekte Abhängigkeit zwischen diesen Variablen entlang des Pfades über* `MAINBOARD_SOCKEL_TYP` *und* `PROZESSOR_SOCKEL_TYP`*. Eine Wertekombination für* `GEHAEUSE_SOCKEL_TYP` *und* `KUEHLER_SOCKEL_TYP` *ist nur dann zulässig, wenn sie ebenfalls für alle Pfade zwischen diesen beiden Variablen zulässig ist, d. h. alle unären und binären Constraints erfüllt sind.*

Allgemein formuliert: Ein Pfad $P = (v_1, \ldots, v_i, \ldots, v_m)$ in einem Constraint-Graph ist pfadkonsistent, wenn sich für alle Wertepaare für (v_1, v_m), die das binäre Constraint $C_{1,m}$ erfüllen, Werte für die dazwischenliegenden Variablen v_i finden lassen, so dass die entsprechenden Constraints $C_{1,2}, \ldots, C_{i,i+1}, \ldots, C_{m-1,m}$ entlang des Pfades erfüllt sind (siehe Abbildung 5.8 auf der gegenüberliegenden Seite). Formal kann dies wie folgt ausgedrückt werden (vgl. Mackworth 1977a, S. 103):

Definition 5.2.9 (Pfadkonsistenz/path consistency, PC)
Gegeben sei ein binäres CSP auf den Variablen $v_1, \ldots, v_n$ mit den Wertebereichen $D_1, \ldots, D_n$. Sei $C_1^u, \ldots, C_n^u$ die Menge der unären Constraints im Constraint-Netz, wobei C_i^u, $i \in \{1, \ldots, n\}$, die Relation R_i^u besitzt und die Variable v_i einschränkt. Weiterhin sei $C_{1,1}^b, \ldots, C_{n,n}^b$ die Menge der binären Constraints im Constraint-Netz, wobei $C_{i,j}^b$, $i, j \in \{1, \ldots, n\}$ und $i \neq j$, die Relation $R_{i,j}^b$ besitzt und die Variablen v_i und v_j einschränkt. Ein Pfad der Länge m mit $m \leq n$ durch die Knoten $v_1, v_2, \ldots, v_{m-1}, v_m$ ist pfadkonsistent, gdw. es für Werte $d_1 \in D_1$ und $d_m \in D_m$, mit $D_1 \subseteq R_1^u$, und $D_m \subseteq R_m^u$

[23]Ein Knoten kann dabei durchaus mehr als einmal in einem Pfad vorkommen, auch unmittelbar hintereinander (vgl. Montanari 1974, S. 109).

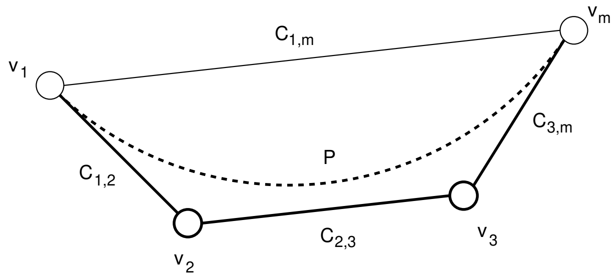


Abbildung 5.8: Pfadkonsistenz

und $(d_1, d_m) \in R^b_{1,m}$*, eine Sequenz von Werten* $d_2 \in D_2, \ldots, d_{m-1} \in D_{m-1}$ *gibt, so dass* $D_2 \subseteq R^u_2, \ldots, D_{m-1} \subseteq R^u_{m-1}$ *und* $(d_1, d_2) \in R^b_{1,2}, \ldots, (d_{m-1}, d_m) \in R^b_{m-1,m}$*.*

Ein Constraint-Netz ist pfadkonsistent, wenn alle Pfade im Constraint-Graph pfadkonsistent sind. Zu beachten ist, dass die Definition von Pfadkonsistenz für einen Pfad $(v_1, \ldots, v_i, \ldots, v_m)$ nicht bedingt, dass konsistente Wertebelegungen für diese Variablen konsistent sind mit Constraints, die nicht Teil dieses Pfades sind. Es wird also wiederum lediglich eine lokale Konsistenz erreicht.

In (Montanari 1974) weist *Ugo Montanari* nach, dass Pfadkonsistenz für *vollständige* Constraint-Graphen erreicht werden kann, indem sichergestellt wird, dass alle Pfade der Länge zwei pfadkonsistent sind (vgl. Mackworth 1977a, S. 111). Es reicht daher aus, innerhalb eines vollständigen Constraint-Netzes die Konsistenz für Wertetripel, d. h. für alle Pfade der Länge zwei, sicherzustellen, damit das gesamte CSP pfadkonsistent ist.[24] Verkürzt lässt sich daher Pfadkonsistenz auch wie folgt definieren (vgl. Mackworth 1977a, S. 100 f.):

$$\begin{aligned} \forall v_i, v_j &: D_i \subseteq R^u_i \wedge D_j \subseteq R^u_j \wedge (d_i, d_j) \in R^b_{i,j} \\ \exists v_k &: D_k \subseteq R^u_k \wedge (d_i, d_k) \in R^b_{i,k} \wedge (d_k, d_j) \in R^b_{k,j} \end{aligned}$$

Infolgedessen arbeiten Algorithmen zur Herstellung von Pfadkonsistenz ausschließlich mit Wertetripeln, d. h. Pfaden der Länge zwei. Sie sind generell als eine Verallgemeinerung der Kantenkonsistenz-Algorithmen anzusehen. Ähnlich wie diese stellen sie Konsistenz her, indem wiederholt Einschränkungen der Wertebereiche der Constraint-Variablen vorgenommen werden. Außerdem nehmen die Algorithmen eine Umformung des vorhandenen Constraint-Netzes vor, indem abgeleitete Constraints, die restriktiver sind und mehr Werte ausschließen als die ursprünglichen Constraints, eingefügt werden bzw. bestehende Constraints überschreiben, bis ein vollständiges, pfadkonsistentes Constraint-Netz vorliegt. Das bestehende Constraint-Netz wird dadurch in ein äquivalentes, pfadkonsistentes Constraint-Netz mit vollständiger Vernetzung überführt.

[24] Auch wenn man eigentlich von einem pfadkonsistenten Constraint-Netz nur sprechen sollte, wenn es sich um ein *vollständiges* Netz handelt, wird in der Literatur Pfadkonsistenz häufig mit (strenger) 3-Konsistenz (vgl. Abschnitt 5.2.3.5, S. 103) gleichgesetzt.

Zur Repräsentation der Abhängigkeiten nutzen Pfadkonsistenz-Algorithmen eine extensionale Darstellung in Form von booleschen Matrizen. Voraussetzung hierfür ist, dass für die Wertebereiche der Constraint-Variablen eine fixe Reihenfolge angenommen wird. Binäre Constraints lassen sich mittels Matrizen darstellen, indem jeder Eintrag einer Matrix für eine konsistente oder inkonsistente Wertekombination der beiden beteiligten Variablen steht. Ein Constraint $C_{i,j}$, welches die Ungleichheit von $v_i \neq v_j$ spezifiziert, wobei die beiden identischen Domänen $D_i = \{1,2\}$ und $D_j = \{1,2\}$ jeweils nur zwei Werte enthalten, wird z. B. durch die folgende Matrix repräsentiert:

$$C_{i,j} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \approx \qquad \left[\begin{array}{c|cc} & 1 & 2 \\ \hline 1 & 0 & 1 \\ 2 & 1 & 0 \end{array} \right]$$

Die Zeilen stehen jeweils für Werte von v_i, die Spalten entsprechend für Werte von v_j. Als Ergebnis steht oben links und unten rechts in der Matrix eine 0, da hier jeweils die beiden ersten bzw. zweiten Werte aus den beiden identischen Domänen D_i und D_j mit $1 \neq 1$ bzw. $2 \neq 2$ eine nach $C_{i,j}$ ungültige Kombination bilden. Entsprechend stellen der erste Wert aus D_i und der der zweite Wert aus D_j oben rechts bzw. der zweite Wert aus D_i und der erste Wert aus D_j unten links mit $1 \neq 2$ und $2 \neq 2$ eine gültige Kombination dar, welches mit einer 1 in der Matrix gekennzeichnet wird.

Aus Gründen der Einheitlichkeit und Vereinfachung werden die Domäne einer Variablen und unäre Constraints in dieser Repräsentation als binäres Constraint $C_{i,i}$ dargestellt, wobei dazu auf der Diagonalen der Matrix jeweils der Wert 1 eingetragen wird. Der Wertebereich $D_i = \{3,4,5\}$ einer Variable v_i wird demnach z. B. durch die folgende Matrix repräsentiert:

$$C_{i,i} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \approx \qquad \left[\begin{array}{c|ccc} & 3 & 4 & 5 \\ \hline 3 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{array} \right]$$

Ein unäres Constraint $C_{i,i}$, welches den Wert 5 für die obige Variable v_i ausschließt, würde durch die folgende Matrix spezifiziert werden:

$$C_{i,i} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \approx \qquad \left[\begin{array}{c|ccc} & 3 & 4 & 5 \\ \hline 3 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 \end{array} \right]$$

Die Arbeitsweise von Pfadkonsistenz-Algorithmen stützt sich auf die folgende Beobachtung: Angenommen das Constraint $C_{1,2}$ beschränkt die Variablen v_1 und v_2 und das Constraint $C_{2,3}$ die Variablen v_2 und v_3. Es lässt sich nun ein Constraint

$$C_{1,3} = C_{1,2} \cdot C_{2,3}$$

ableiten (engl. *composite relation*), welches sich durch die binäre Multiplikation der beiden Matrizen $C_{1,2}$ und $C_{2,3}$ ergibt.[25] Dieser Mechanismus wird engl. ***composition*** genannt und ist wie folgt definiert (vgl. Tsang 1993, S. 91):

$$\begin{aligned} C_{i,j} &= C_{i,k} \cdot C_{k,j}, \;\; gdw. \\ C_{i,j,r,s} &= (C_{i,k,r,1} \wedge C_{k,j,1,s}) \vee (C_{i,k,r,2} \wedge C_{k,j,2,s}) \vee \ldots \vee (C_{i,k,r,t} \wedge C_{k,j,t,s}) \end{aligned}$$

Wobei r die r-te Zeile, s die s-te Spalte und t die Kardinalität von D_y bezeichnet. Ein Beispiel für eine derartige Operation:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Ein Wertepaar (a,c) der Variablen v_1 und v_3, ist mit $C_{1,3}$ nur dann konsistent, wenn es ebenfalls ein konsistentes Wertepaar (a,b) für v_1 und v_2 mit $C_{1,2}$ und ein konsistentes Paar (b,c) für v_2 und v_3 mit $C_{2,3}$ gibt. Dies lässt sich darauf verallgemeinern, dass die Werte, die v_i und v_j gleichzeitig als Teil einer Lösung annehmen können, außer durch das Constraint $C_{i,j}$ ebenfalls durch die Verknüpfung aller Constraints $C_{i,k} \cdot C_{k,k} \cdot C_{k,j}$ für alle $v_k \in V$ eingeschränkt werden (vgl. Tsang 1993, S. 92):

$$C_{i,j} = C_{i,j} \wedge C_{i,k_1} \cdot C_{k_1,k_1} \cdot C_{k_1,j} \wedge C_{i,k_2} \cdot C_{k_2,k_2} \cdot C_{k_2,j} \wedge \ldots \wedge C_{i,k_n} \cdot C_{k_n,k_n} \cdot C_{k_n,j}$$

Wobei n die Anzahl der Variablen des CSP ist. Der einfachste, erste Pfadkonsistenz-Algorithmus `PC-1` in Abbildung 5.9 auf der nächsten Seite macht sich dies zu Nutze, indem er für jede Variable v_k mit $k \in \{1,\ldots,n\}$ des CSP jedes Constraint $C_{i,j}$ mittels *composition* reduziert:[26]

$$Y_{i,j}^{k} \leftarrow Y_{i,j}^{k-1} \;\&\; Y_{i,k}^{k-1} \cdot Y_{k,k}^{k-1} \cdot Y_{k,j}^{k-1}$$

Y^k steht dabei jeweils für eine aktuelle Menge von Constraints.[27] $Y_{i,j}^{k}$ benennt das Constraint $C_{i,j}$ aus der Menge Y^k. Y^k wird jeweils dazu genutzt, den Nachfolger Y^{k+1} zu erstellen. Der Durchlauf durch alle Schleifen des Algorithmus wird ähnlich wie bei `AC-1` so oft wiederholt, bis keine Änderung an einem Constraint mehr vorgenommen wird ($Y^n = Y^0$). Der Algorithmus iteriert für alle Variablen v_k durch alle Constraints $C_{i,j}$, welche durch Nutzung des *composition*-Operators aus $C_{i,k}$, $C_{k,k}$ und $C_{k,j}$ neu berechnet werden. Die &-Operation kennzeichnet die logische UND-Verknüpfung zweier Matrizen und stellt sicher, dass die Restriktionen eines bestehenden Constraints $C_{i,j}$ berücksichtigt werden (engl. ***intersection***).[28]

[25] Die Multiplikation entspricht dabei dem logischen UND, die Addition dem logischen OR (vgl. Schöning 2000).

[26] Der Algorithmus `PC-1` wurde ursprünglich von Montanari (1974, S. 113) mit *„Algorithm C“* benannt (*„C“* steht in diesem Fall für engl. *closure*, womit i. A. das Erreichen eines bestimmten Konsistenzgrades gemeint ist).

[27] C ist die Menge aller Constraints des CSP (vgl. Definition 5.2.1, S. 83).

[28] Das Ergebnis ist die Schnittmenge der beiden Matrizen, wobei die *composition*-Operation vorrangig vor der UND-Verknüpfung ist und eine höhere Bindungsstärke aufweist. Die logische UND-Verknüpfung zweier Matrizen garantiert, dass sich in der Ergebnismatrix nur konsistente Wertekombinationen befinden, die in beiden Constraints bzw. Matrizen erlaubt sind.

begin
  $Y^n \leftarrow C$
  repeat
    begin
      $Y^0 \leftarrow Y^n$
        for $k \leftarrow 1$ **until** n **do**
          for $i \leftarrow 1$ **until** n **do**
            for $j \leftarrow 1$ **until** n **do**
              $Y^k_{i,j} \leftarrow Y^{k-1}_{i,j}$ & $Y^{k-1}_{i,k} \cdot Y^{k-1}_{k,k} \cdot Y^{k-1}_{k,j}$
    end
  until $Y^n = Y^0$;
  $Y \leftarrow Y^n$
end

Abbildung 5.9: Der Pfadkonsistenz-Algorithmus `PC-1` (vgl. Mackworth 1977a, S. 111)

Wie `AC-1` ist der Algorithmus `PC-1` sehr ineffizient, da für jede Änderung an nur einem einzigen Constraint der Algorithmus komplett erneut durchlaufen werden muss. Aufgrund der vielen Variablen Y^k ist `PC-1` zudem sehr speicherintensiv. Der Algorithmus `PC-2` (vgl. Mackworth 1977a, S. 113) ist ähnlich wie `AC-3` eine Verbesserung in der Hinsicht, dass bei Änderungen nur noch die betroffenen Constraints erneut geprüft werden müssen.

Neben `PC-1` und `PC-2` gibt es noch eine Reihe weiterer, bekannter Pfadkonsistenz-Algorithmen. Sie sind jeweils Verallgemeinerungen der entsprechenden Algorithmen zur Herstellung von Kantenkonsistenz. So ist der Algorithmus `PC-3` eine Verbesserung von `PC-2`, der seine erhöhte Effizienz mit ähnlichen Mechanismen und denselben Nachteilen bzgl. erhöhtem Speicherbedarf zur Verwaltung von zusätzlichen Informationen erreicht, wie `AC-4` gegenüber `AC-3`. `PC-3` wird zusammen mit `AC-4` von Mohr und Henderson (1986) vorgestellt. Aufgrund kleinerer Fehler in `PC-3` präsentierten *Ching-Chih Han* und *Chia-Hoang Lee* eine korrigierte Version namens `PC-4` (vgl. Han und Lee 1988).[29] Die Algorithmen `PC-5` und `PC-5++` von *Moninder Singh* sind weitere Verbesserungen, basierend auf `AC-6` bzw. `AC-6++` (vgl. Singh 1995). Hauptvorteil ist die geringere Platzkomplexität der Algorithmen, wobei die *worst-case* Zeitkomplexität gleich der von `PC-4` ist. Das durchschnittliche Laufzeitverhalten ist jedoch deutlich verbessert.

Unabhängig von `PC-5` wurde der Algorithmus `PC-6` entwickelt, der ebenfalls das Prinzip des *minimal support* von `AC-6` auf die Pfadkonsistenz überträgt (vgl. Chmeiss 1996; Chmeiss und Jégou 1995)[30]. Die Algorithmen bzw. deren Eigenschaften sind demnach identisch. Die Algorithmen `PC-7` (vgl. Chmeiss und Jégou 1996a) und `PC-8` (vgl. Chmeiss und Jégou 1996b, 1998) von *Assef Chmeiss* und *Philippe Jégou* basieren ebenfalls wie `PC-5` bzw. `PC-6` darauf, nicht benötigte Konsistenzberechnungen durch *minimal support* zu vermeiden, allerdings ohne diese *supports* in einer aufwendigen Datenstruktur zu speichern. Der Algorithmus `PC-8` ist dabei als eine Optimierung von `PC-7` hinsichtlich der Platzkom-

[29] `PC-3` ist inkorrekt und entfernt u. U. konsistente Werte aus den Domänen der Constraint-Variablen.

[30] Zit. nach Chmeiss und Jégou (1996a, S. 196); Chmeiss und Jégou (1996b, S. 286); Chmeiss und Jégou (1998, S. 122).

plexität zu sehen. Einer Verbesserung des Speicherbedarfs steht bei beiden Algorithmen eine leichte Verschlechterung des Laufzeitverhaltens gegenüber. Eine Übersicht über die jeweiligen *worst-case* Zeit- und Platzkomplexitäten der hier angesprochenen Algorithmen befindet sich in der Tabelle 5.1 auf Seite 106.

Obwohl die Pfadkonsistenz einen höheren Konsistenzgrad darstellt als die Kantenkonsistenz und ggf. wesentlich mehr Werte aus den Wertebereichen der Variablen eines CSP entfernt, die nicht Teil einer Lösung sein können, werden die entsprechenden Algorithmen zum Herstellen von Pfadkonsistenz, außer bei speziellen Problemstellungen, relativ selten eingesetzt. Dies liegt darin begründet, dass Pfadkonsistenz mehrere Probleme mit sich bringt (vgl. Barták 2001, S. 10): So filtern Pfadkonsistenz-Algorithmen zwar mehr inkonsistente Werte aus den Wertebereichen heraus, allerdings wird dafür im Vergleich zur Kantenkonsistenz ein wesentlich schlechteres Verhältnis in Bezug zur Ausführungsgeschwindigkeit bzw. Zeitkomplexität in Kauf genommen. Weiterhin ist der Speicherbedarf von Pfadkonsistenz-Algorithmen durch die extensionale Repräsentation aller Constraints bzw. Domänen auch für überschaubare Problemstellungen wesentlicher höher als bei Verfahren zur Herstellung von Kantenkonsistenz. Zudem werden bei der Herstellung von Pfadkonsistenz abgeleitete Constraints zu dem Constraint-Netz hinzugefügt. Suchverfahren zur Bestimmung von Lösungen, die bestimmte, vormals vorhandene Strukturen eines Constraint-Netzes ausnutzen, lassen sich dadurch nicht mehr anwenden. Und letztendlich stellt Pfadkonsistenz i. A. immer noch lediglich lokale Konsistenz her, d. h. es werden nicht alle inkonsistenten Werte aus den Domänen entfernt.

Eine besondere Rolle bzgl. Pfadkonsistenz spielen allerdings sog. *row-convex*-Constraints. Ein binäres Constraint in Matrixrepräsentation ist *row-convex*, wenn in jeder Zeile der Matrix die 1er Einträge aufeinander folgend sind und nicht durch eine 0 in derselben Zeile unterbrochen werden. Für diese Art Constraints wird von van Beek (1992) nachgewiesen, dass Pfadkonsistenz gleichbedeutend mit globaler Konsistenz ist. Die Konvexitätseigenschaften haben eine besondere Relevanz z. B. für Zeitplanungsprobleme formalisiert durch TCSPs (vgl. Abschnitt 4.4.1, S. 57) und bei Verfahren zum Auffinden von Lösungen in ICSPs (vgl. Abschnitt 5.3.1.4, S. 149 und Abschnitt 5.3.7, S. 167).

5.2.3.5 k-Konsistenz

Das Konzept der Knoten-, Kanten- und Pfadkonsistenz wurde von *Eugene C. Freuder* um den Begriff der *k-Konsistenz* erweitert (vgl. Freuder 1978). Die k-Konsistenz ist eine Verallgemeinerung der bisher genannten Konsistenzgrade und garantiert, dass in einem Constraint-Netz für jede Menge von (k-1)-konsistent belegten Variablen eine konsistente Belegung der k-ten Variable existiert (vgl. Güsgen 2000, S. 270):

Definition 5.2.10 (k-Konsistenz/k-consistency)
Gegeben sei ein beliebiges CSP auf den Variablen $v_1, \ldots, v_n$ mit den Wertebereichen $D_1, \ldots, D_n$. Sei $(d_{i_1}, \ldots, d_{i_{(k-1)}}) \in D_{i_1} \times \cdots \times D_{i_{(k-1)}}$ mit $i_j \in \{1, \ldots, n\}$ für $j = \{1, \ldots, k\}$ eine Zuweisung von Werten an $(k-1)$ paarweise verschiedene Variablen, die alle Constraints zwischen den Variablen erfüllt. Das Netz ist k-konsistent, wenn durch Hinzunahme einer beliebigen weiteren Variablen $v_{i_k}, k \in \{1, \ldots, n\}$, die Wertezuweisung zu

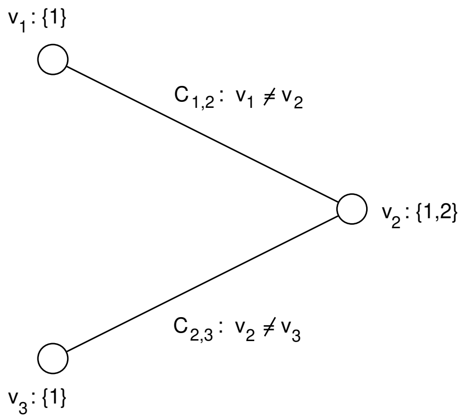


Abbildung 5.10: 3-konsistentes, jedoch nicht 2-konsistentes Constraint-Netz

$(d_{i_1}, \ldots, d_{i_k}) \in D_i \times \cdots \times D_{i_k}$ *so erweitert werden kann, dass alle Constraints zwischen den k Variablen erfüllt sind.*

Es ist wichtig festzustellen, dass k-Konsistenz nicht automatisch impliziert, dass ein Constraint-Netz ebenfalls (k-1)-konsistent ist. Ein einfaches Beispiel nach Freuder (1982, S. 26 f.) ist in Abbildung 5.10 zu sehen:

Beispiel 5.2.3 *Das dargestellte Constraint-Netz in Abbildung 5.10 ist 3-konsistent aber nicht 2-konsistent. 2-Konsistenz ist deshalb ausgeschlossen, weil es für die Belegung* $v_2 = 1$ *weder eine konsistente Belegung für* v_1 *noch für* v_3 *gibt, so dass die Constraints* $C_{1,2}$ *und* $C_{2,3}$ *erfüllt würden. 3-Konsistenz hingegen ist sichergestellt, da sich, wie von der Definition gefordert, alle konsistenten Belegungen von zwei Variablen um eine dritte Variable mit konsistenter Belegung erweitern lassen:* $v_1 = 1, v_2 = 2, v_3 = 1$.

Abhilfe schafft in diesem Fall der von Freuder (1982) eingeführte Begriff der *strengen k-Konsistenz* (vgl. Güsgen 2000, S. 271):

Definition 5.2.11 (strenge *k*-Konsistenz/strong *k*-consistency)
Ein Constraint-Netz ist streng k-konsistent, wenn es i-konsistent ist für alle $1 \leq i \leq k$.

Nach diesen Definitionen ist Knotenkonsistenz äquivalent zur 1-Konsistenz, Kantenkonsistenz äquivalent zur strengen 2-Konsistenz und Pfadkonsistenz äquivalent zur strengen 3-Konsistenz (vgl. Barták 1999a, S. 558; Barták 1999b, S. 10). Wenn ein CSP mit n Variablen streng n-konsistent ist, sind die Wertebereiche aller Constraint-Variablen minimal, d. h. sie können nicht weiter eingeschränkt werden, ohne dass mögliche Lösungen verloren gehen. In diesem Zustand ist ein CSP *global konsistent*. Globale Konsistenz garantiert Konfliktfreiheit und erlaubt dadurch die Generierung einer konsistenten Lösung in linearer Zeit. Die Wertebereiche der Constraint-Variablen enthalten ausschließlich Werte, die Teil einer Lösung sind. Somit lässt sich jede konsistente Teilbelegung der Constraint-

Variablen zu einer vollständigen Lösung des CSP erweitern, ohne dass Backtracking angewendet werden muss (vgl. Abschnitt 5.2.4.2, S. 114).

Eine interessante Beobachtung geht auf die von Freuder (1982, S. 30 f.) beschriebenen Zusammenhänge zurück. Demnach kann in einem knoten- und kantenkonsistenten, binären CSP,[31] dessen Constraint-Graph einen Baum darstellt und demnach keine Zyklen enthält (engl. *tree-structured*), eine Lösung in linearer Zeit ohne jegliches Backtracking der Suchstrategie gefunden werden (engl. *backtrack-free*).[32] Das heißt in einem azyklischem Constraint-Netz ist diese Form lokaler Konsistenz gleichbedeutend mit globaler Konsistenz. In Abhängigkeit von der Struktur des Constraint-Graphen ist man daher sehr an effizienten Verfahren zur Herstellung lokaler Konsistenz interessiert (vgl. Dechter 1992; Dechter und Pearl 1987, 1989; Mackworth und Freuder 1985).

Allgemein allerdings impliziert k-Konsistenz nicht, dass ein CSP lösbar ist.[33] Auch für ein Constraint-Netz, welches streng k-konsistent ist, mit $k < n$, lässt sich Backtracking bei einem anschließenden Suchverfahren zum Generieren einer Lösung nicht ausschließen, da das Constraint-Netz weiterhin inkonsistente Wertekombinationen enthalten kann. Umgekehrt impliziert eine Lösung nicht k-Konsistenz, d. h. eine Lösung setzt nicht voraus, dass ein Constraint-Netz einen bestimmten Konsistenzgrad aufweist (vgl. Tsang 1993, S. 64 f.). Wie bereits erwähnt, werden Konsistenz- und Suchverfahren häufig kombiniert, um ein CSP effizient zu lösen. Die Effizienz von Suchverfahren kann gesteigert werden, indem Konsistenzverfahren zur Reduzierung des Suchraums zum Einsatz kommen. Allerdings erfordert die Anwendung von Konsistenzverfahren ebenfalls einen nicht unerheblichen Aufwand (siehe Tabelle 5.1, S. 106). Je mehr Aufwand in die Problemreduktion investiert wird, umso weniger Aufwand wird bei einem anschließenden Suchverfahren zur Auffindung von Lösungen des Problems benötigt. In Abbildung 5.11 auf Seite 107 ist grob die Beziehung zwischen dem Aufwand zur Problemreduktion und dem Aufwand zur Lösungssuche skizziert. In der Praxis muss häufig eine Balance zwischen beiden gefunden werden.

Der von Freuder (1978) vorgestellte *Synthese-Algorithmus* ist in der Lage, für beliebige $k \in \{1, \ldots, n\}$, wobei n die Anzahl der Variablen des CSP darstellt, in einem Constraint-Netz strenge k-Konsistenz herzustellen. Dazu konstruiert der Algorithmus nach und nach Constraints mit zunehmender Stelligkeit und fügt diese dem Constraint-Netz hinzu, bis nach k Durchläufen strenge k-Konsistenz erreicht ist. Eine Weiterführung dieses Algorithmus, der ebenfalls k-Konsistenz für beliebige k herstellt, wird von *Martin C. Cooper* vorgestellt. Coopers Algorithmus `KS-1` ist optimal und nimmt Anleihen sowohl bei Freuders Arbeit als auch beim `PC-4`-Algorithmus (vgl. Cooper 1989). Beide Algorithmen sind nicht auf binäre Constraints beschränkt, sondern können auf n-äre Constraint-Netze angewendet werden. Aufgrund des hohen Berechnungsaufwandes unterbleibt dies jedoch häufig in der Praxis.

Synthese-Algorithmen können auch dazu genutzt werden, in einem CSP mit n Variablen strenge n-Konsistenz, d. h. globale Konsistenz, herzustellen (engl. *solution syn-*

[31] DAC ist in diesem Fall ausreichend (vgl. Dechter und Pearl 1987, S. 11; Tsang 1993, S. 63 und Abschnitt 5.2.3.6, S. 107).

[32] Vgl. *minimal width ordering*, Abschnitt 5.2.5.1, S. 129.

[33] Ein kanten- oder pfadkonsistentes Constraint-Netz besitzt nicht zwangsläufig eine Lösung.

e = Anzahl der Constraints d = Kardinalität der größten Domäne n = Anzahl der Variablen		
Algorithmus	**worst-case Zeitkomplexität**	**worst-case Platzkomplexität**
`NC-1` (Mackworth 1977a)	$O(nd)$	$O(nd)$
`AC-1` (Mackworth 1977a)	$O(ned^3)$	$O(e+nd)$
`AC-3` (Mackworth 1977a)	$O(ed^3)$	$O(e+nd)$
`AC-4` (Mohr und Henderson 1986)	$O(ed^2)$	$O(ed^2)$
`AC-5` (Van Hentenryck et al. 1992)	$O(ed)$ (nur für bestimmte Constraint-Klassen)	$O(ed+nd)$
`AC-6` (Bessière und Cordier 1993; Bessière 1994a)	$O(ed^2)$	$O(ed)$
`AC-7` (Bessière et al. 1995, 1999a)	$O(ed^2)$	$O(ed)$
`AC-8` (Chmeiss und Jégou 1996b, 1998)	$O(ed^3)$	$O(n)$
`AC-2000` (Bessière und Régin 2001a)	$O(ed^3)$	$O(nd)$
`AC-2001` (Bessière und Régin 2001a)	$O(ed^2)$	$O(ed)$
`AC-3.1` (Zhang und Yap 2001)	$O(ed^2)$	$O(ed)$
`AC-3`$_d$ (van Dongen 2002a, b)	$O(ed^3)$	$O(e+nd)$
`PC-1` (Mackworth 1977a)	$O(n^5d^5)$	$O(n^3, d^2)$
`PC-2` (Mackworth 1977a)	$O(n^3d^5)$	$O(n^3+n^2d^2)$
`PC-3` (Mohr und Henderson 1986)	$O(n^3d^3)$	$O(n^3d^3)$
`PC-4` (Han und Lee 1988)	$O(n^3d^3)$	$O(n^3d^3)$
`PC-5`/`PC-6` (Chmeiss und Jégou 1995; Chmeiss 1996; Singh 1995)	$O(n^3d^3)$	$O(n^3d^2)$
`PC-7` (Chmeiss und Jégou 1996a)	$O(n^3d^4)$	$O(n^2d^2)$
`PC-8` (Chmeiss und Jégou 1996b, 1998)	$O(n^3d^4)$	$O(n^2d)$
Freuders Synthese-Algorithmus (Freuder 1978)	$O(2^n+nd^{2n})$	$O(2^nd^n)$

Tabelle 5.1: Übersicht über die *worst-case* Zeit- und Platzkomplexitäten

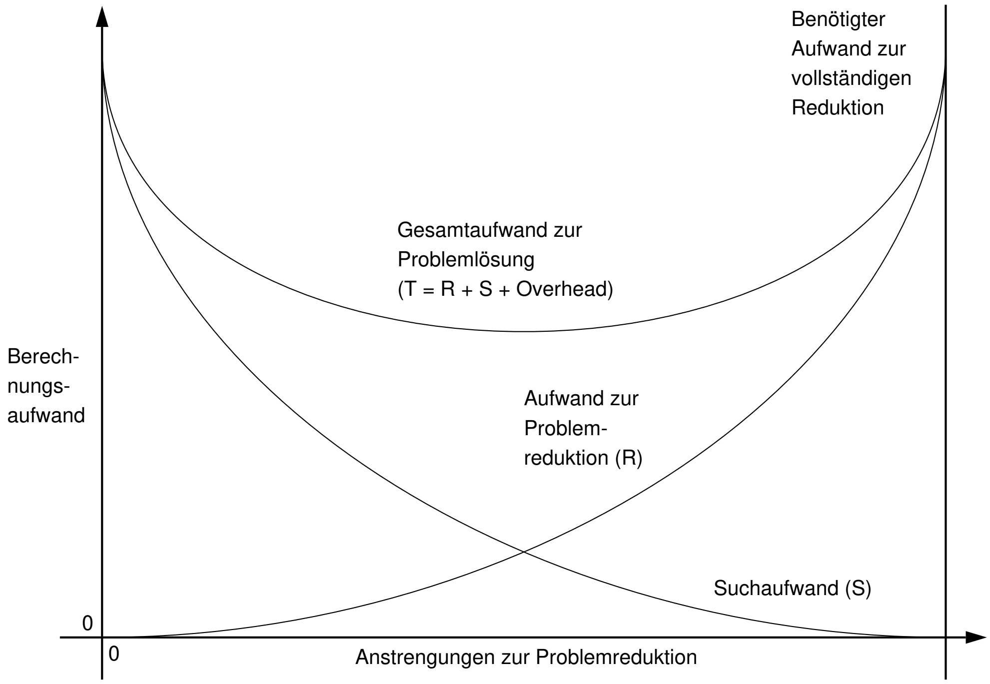


Abbildung 5.11: Aufwand von Problemreduktion vs. Suchaufwand (vgl. Tsang 1993, S. 42)

thesis). So können Synthese-Algorithmen einerseits durch Berechnung von k-Konsistenz, mit $k < n$, als Preprozessor zur Reduktion des Lösungsraums dienen, um ggf. ein nachfolgendes Suchverfahren zu vereinfachen, andererseits können sie genutzt werden, durch Berechnung von n-Konsistenz, alle Lösungen eines CSP zu finden.[34] Allerdings sind Algorithmen, die strenge k-Konsistenz in einem Constraint-Netz herstellen, von exponentieller Komplexität. Grundsätzlich ist die Anwendung von Algorithmen, die einen hohen Konsistenzgrad herstellen und damit einen hohen Komplexitätsgrad aufweisen, immer dann sinnvoll, wenn sichergestellt ist, dass sie eine große Menge redundanter Werte aus den Domänen der Constraint-Variablen eines stark beschränkten Problems entfernen und die Effizienz eines nachfolgenden Suchverfahrens dadurch entsprechend gesteigert wird (vgl. Tsang 1993, S. 136).

5.2.3.6 Weitere Konsistenzverfahren

Neben den bisher vorgestellten Konsistenzgraden und Verfahren, gibt es noch eine Reihe weiterer bekannter Variationen, Kombinationen und Weiterführungen der bisher genannten Verfahren, die in diesem Abschnitt angesprochen werden. In der nachfolgenden, übersichtsartigen Auflistung werden die wesentlichen Merkmale jeweils kurz angesprochen:

[34]Das k-stellige Constraint, welches der Synthese-Algorithmus von Freuder (1978) berechnet, stellt die Menge der Lösungen des CSP dar, so dass die Anwendung eines Suchverfahrens entfällt.

(i,j)-consistency: Auch k-Konsistenz kann noch weiter verallgemeinert werden. In einem *(i,j)-konsistenten* Constraint-Netz können i unterschiedliche, mit konsistenten Werten belegte Variablen um j weitere Variablen zur einer konsistenten Belegung erweitert werden (vgl. Freuder 1985, S. 757). k-Konsistenz ist demnach äquivalent zu (k-1, 1)-Konsistenz. Für $k = 2$ entspricht $(1,1)$-Konsistenz der Kantenkonsistenz und für $k = 3$ entspricht $(2,1)$-Konsistenz der Pfadkonsistenz. Strenge (i,j)-Konsistenz ist gegeben, wenn ein Constraint-Netz (k,j)-konsistent ist, für alle $k \leq i$. Algorithmen, die (i,j)-Konsistenz herstellen können, besitzen eine hohe Platzkomplexität, da sie Tupel mit i Werten verwalten müssen. Sie sind daher praktisch häufig nicht nutzbar für $i \geq 2$. Zudem belassen sie das Constraint-Netz nicht in seinem ursprünglichen Zustand, sondern überführen es in ein äquivalentes Constraint-Netz (vgl. Barták 2001, S. 11).

inverse consistency: Da beim Erhöhen von i die effiziente Anwendbarkeit von (i,j)-Konsistenz stark nachlässt, wird bei der *inversen Konsistenz* $i = 1$ belassen und stattdessen j erhöht. Dementsprechend wird (1, k-1)-Konsistenz *k-inverse Konsistenz* bzw. engl. *k-inverse consistency* genannt (vgl. Freuder und Elfe 1996). (1, k-1)-Konsistenz bedeutet, dass durch diesen Konsistenzgrad alle Werte entfernt werden, die nicht um (k-1)-konsistente Variablenbelegungen erweitert werden können (vgl. Freuder 1985). *Inverse Kantenkonsistenz* ist äquivalent zur gewöhnlichen Kantenkonsistenz (d. h. $(1,1)$-Konsistenz). *Inverse Pfadkonsistenz* (engl. *path inverse consistency*, PIC) ist eine inverse Konsistenz, die mehr inkonsistente Werte eliminiert als Kantenkonsistenz, dabei allerdings nicht den Konsistenzgrad der vollständigen Pfadkonsistenz erreicht. Der (inverse) Konsistenzgrad erhöht sich mit der Größe von k, allerdings steigt entsprechend auch die Zeitkomplexität polynomial mit k. Daher ist auch inverse Konsistenz für große k nur beschränkt anwendbar. Ein optimaler, auf `AC-7` basierender Algorithmus `PIC-2` wird von Debruyne (2000) vorgestellt. Eine Variante ist die *neighborhood inverse consistency* (NIC), die sicherstellt, dass der Wert einer Variable um eine konsistente Belegung aller unmittelbaren Nachbarn der Variable erweitert werden kann (vgl. Freuder und Elfe 1996). Allerdings ist *worst-case* Zeitkomplexität von NIC exponentiell. So würde in einem vollständigen Constraint-Netz durch dieses Verfahren globale Konsistenz hergestellt werden. Die Anwendung von NIC sollte sich daher auf überschaubare, schwach beschränkte Constraint-Netze begrenzen.

lazy arc consistency (LAC): Einen Sonderfall stellt die *lazy arc consistency* dar (vgl. Schiex et al. 1996): Im Gegensatz zur gewöhnlichen Kantenkonsistenz, die die Domänen $D = \{D_1, \ldots, D_n\}$ der Constraint-Variablen um alle kanteninkonsistenten Werte reduziert, ist das Ziel von LAC das Erkennen von Inkonsistenzen durch Herbeiführen eines *domain wipe out*, d. h. der Wertebereich einer Variablen enthält keine Elemente mehr. Während Kantenkonsistenz für gewöhnlich eine maximale, kantenkonsistente Teildomäne $D' = \{D'_1, \ldots, D'_n\}$ zurückliefert, ist es bei LAC ausreichend, wenn eine beliebige kantenkonsistente Teildomäne gefunden wird, die lediglich eine konsistente Belegung enthalten muss, weil dadurch nachgewiesen ist, dass ein *domain wipe out* nicht auftreten kann. Der von Schiex et al. (1996) vorgestellte Algorithmus $\texttt{LAC}_7$

sowie die beiden Varianten `LAC`$_7^+$ und `MinLAC`$^+$$+_7$ sind modifizierte Versionen des `AC-7`-Algorithmus.

directional arc consistency (DAC): *Rina Dechter* und *Judea Pearl* stellen fest, dass es für baumartige Constraint-Netze, die keine Zyklen enthalten, nicht notwendig ist, vollständige Kantenkonsistenz herzustellen, um eine Backtracking-freie Suchstrategie zu ermöglichen (vgl. Dechter und Pearl 1987, S. 10 ff.). Algorithmen zur Herstellung von Kantenkonsistenz müssen, wie in Abschnitt 5.2.3.3 auf Seite 95 angemerkt, den `REVISE`-Algorithmus mehrfach auf einzelne Kanten anwenden, da die Reduktion des Wertebereichs einer Variablen aufgrund von Zyklen im Constraint-Netz Auswirkungen auf Wertebereiche von vorher bereits eingeschränkten Variablen haben kann. Algorithmen, die `REVISE` ausschließlich auf geordnete Variablenpaare (v_i, v_j) mit $i < j$ anwenden, stellen einen Konsistenzgrad her, der *gerichtete Kantenkonsistenz* (engl. *directional arc consistency*) genannt wird. Gerichtete Kantenkonsistenz ist schwächer als vollständige Kantenkonsistenz lässt sich aber erheblich effizienter, d. h. mit weniger Berechnungsaufwand als `AC-3` und weniger Speicherbedarf als `AC-4`, herstellen.[35] Eine geschickte Anordnung der Variablen kann dabei die Reduzierung der Domänen erhöhen.

directional path consistency (DPC): Ähnlich wie DAC für AC existiert eine abgeschwächte Form der Pfadkonsistenz, die *gerichtete Pfadkonsistenz* (engl. *directional path consistency*). Gerichtete Pfadkonsistenz nutzt wie DAC eine geordnete Variablenliste. Es werden grundsätzlich dieselben Umformungen vorgenommen, wie bei normalen Pfadkonsistenz-Algorithmen, allerdings werden nur ausgewählte Relationen mit $v_i < v_j < v_k$ Konsistenztests unterzogen. Das Ergebnis ist ein Konsistenzgrad, der weniger Berechnungsaufwand erfordert als vollständige Pfadkonsistenz, dabei die Domänen der Constraint-Variablen allerdings um nicht ganz so viele inkonsistente Wertekombinationen reduziert (vgl. Dechter und Pearl 1987, S. 14 ff.).

restricted path consistency (RPC): Aufgrund der dargelegten Nachteile wird Pfadkonsistenz relativ selten eingesetzt. Eine schwächere Form, die eine Kombination aus Pfad- und Kantenkonsistenz darstellt, ist die *eingeschränkte Pfadkonsistenz* (engl. *restricted path consistency*). Der `RPC-1`-Algorithmus von Berlandier (1995) basiert auf `AC-4`, erweitert um Filtermethoden der Pfadkonsistenz, die immer dann angewendet werden, wenn es für einen Wert einer Variable in einem (binären) Constraint nur eine einzige konsistente Wertekombination gibt, die das Constraint erfüllt (durch einen entsprechenden *support*-Eintrag, vgl. Abschnitt 5.2.3.3, S. 96). Ist diese Wertekombination inkonsistent mit den übrigen Constraints, kann der ursprüngliche Wert aus seiner Domäne gelöscht werden. Im Gegensatz zu anderen Pfadkonsistenz-Algorithmen ändert RPC nur die Domänen der Constraint-Variablen, nicht hingegen das Constraint-Netz an sich durch Hinzufügen neuer Constraints. RPC ist schwächer

[35] Anders als von Dechter und Pearl (1987, S. 13) hervorgehoben garantiert allerdings die zweimalige Anwendung des DAC-Algorithmus (Hin- und Rückrichtung) auf einem Constraint-Graphen, der einen Baum darstellt, *nicht* vollständige Kantenkonsistenz (vgl. Tsang 1993, S. 89 f.; Tsang 1998, S. 356).

als vollständige Pfadkonsistenz, reduziert die Domänen der Constraint-Variablen jedoch um mehr Werte als Kantenkonsistenz. Ein verbesserter Algorithmus `RPC-2` und die Erweiterungen k-RPC und Max-RPC, die jeweils eigene, höhere Konsistenzgrade darstellen, werden von Debruyne und Bessière (1997a) bzw. Debruyne (1998) vorgestellt.[36]

singleton consistency: Verfahren zum Herstellen von *Singleton-Konsistenz* sind generisch und können mit beliebigen Konsistenzverfahren kombiniert werden, um die Reduktion der Wertebereiche der Constraint-Variablen zu erhöhen (vgl. Debruyne und Bessière 1997b; Prosser et al. 2000). Zum Erreichen von Singleton-Konsistenz wird nacheinander für alle Variablen geprüft, ob das Beschränken der jeweiligen Domäne auf ein einziges Element (engl. *singleton*) durch Anwenden von Konsistenztechniken zu einem inkonsistenten CSP, d. h. zu min. einem leeren Wertebereich, führt. Ist dies der Fall, kann der entsprechende Wert aus der Domäne gelöscht werden. Die Überprüfung muss jeweils für alle möglichen Werte einer Variablen durchgeführt werden. Zur Konsistenzüberprüfung sollte aus Komplexitätsgründen eine lokale Konsistenz hergestellt werden, z. B. Kantenkonsistenz (*singleton arc consistency*, SAC)[37] oder beschränkte Pfadkonsistenz (*singleton restricted path consistency*, SRPC). Wenn lokale Konsistenz mit polynomialen Zeitaufwand hergestellt werden kann, besitzt die entsprechende Singleton-Konsistenz ebenfalls polynomiale *worst-case* Zeitkomplexität.

adaptive consistency: *Adaptive Konsistenz* ist eine gerichtete, globale Konsistenz bezogen auf eine bestimmte Anordnung der Variablen, d. h. es wird globale Konsistenz für eine gegebene Ordnung der Variablen hergestellt (vgl. Dechter und Pearl 1987, S. 17 ff.). Das Verfahren wird *adaptiv* genannt, weil die Eigenschaften des Constraint-Graphen die anzuwendenden Konsistenztechniken beeinflussen. Die Variablen werden in der gegebenen Ordnung absteigend bearbeitet. Es wird dabei Konsistenz mit den jeweils übergeordneten Variablen hergestellt, d. h. für eine Variable mit einem Vorgänger wird Kantenkonsistenz hergestellt, bei zwei Vorgängern entsprechend Pfadkonsistenz usw. Das Ergebnis ist ein geordneter Constraint-Graph, aus dem Lösungen mittels systematischer Suche ohne Anwendung von Backtracking generiert werden können. Allerdings wird hierfür exponentielle Zeit- und Platzkomplexität in Anspruch genommen. Zudem wird bei Anwendung entsprechender Konsistenzverfahren (Pfadkonsistenz) der Constraint-Graph modifiziert.

Diese Auflistung erhebt keinen Anspruch auf Vollständigkeit. Sie zeigt allerdings auf, wie „reichhaltig“ die Möglichkeiten des Preprozessings für CSPs durch Konsistenztechniken sind. Zum Teil setzen die genannten Verfahren spezielle CSPs bzw. spezielle Strukturen voraus. Wie bereits erwähnt werden in der Praxis trotz der vielen Varianten häufig generelle und einfach zu implementierende Techniken, wie die Kantenkonsistenz, eingesetzt, die unabhängig von der Struktur eines Problems mit geringer Komplexität bereits sehr viele

[36] Anstatt Pfadkonsistenz nur dann herzustellen, wenn es nur eine kompatible Wertekombination gibt, kann dies bereits durchgeführt werden, wenn k kompatible Wertekombinationen vorhanden sind.

[37] Wobei sich LAC in diesem Fall anbietet.

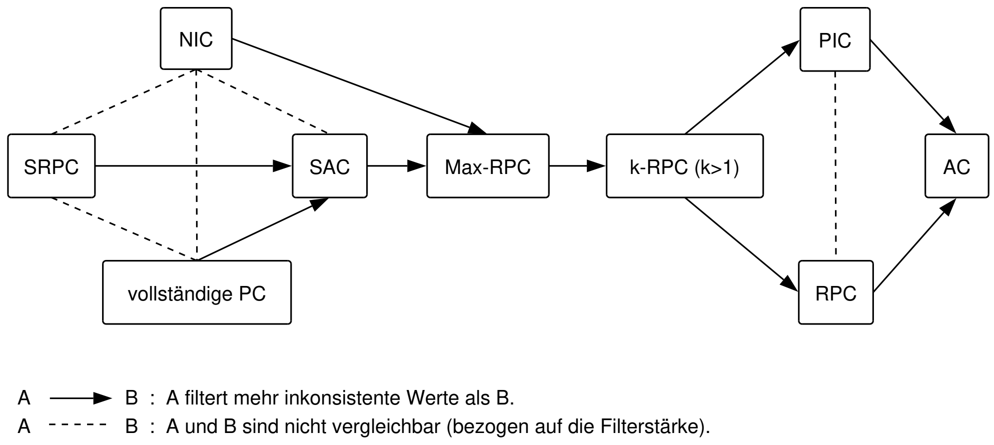


Abbildung 5.12: Filterstärke unterschiedlicher Konsistenztechniken (vgl. Debruyne und Bessière 2001, S. 217)

Inkonsistenzen entdecken und herausfiltern können. Von besonderem Interesse sind daher Verfahren, die stärker als Kantenkonsistenz sind, dabei aber im Gegensatz zur Pfadkonsistenz das Constraint-Netz nicht verändern (z. B. *restricted path consistency* und *singleton consistency*) und mit einem vertretbaren Aufwand anwendbar sind.[38] Eine Übersicht (siehe Abbildung 5.12) und formale Vergleiche bzgl. der Filterstärke von praxisrelevanten Konsistenztechniken sind den Arbeiten von Debruyne und Bessière (1997b, 2001) und Prosser et al. (2000) zu entnehmen.

5.2.4 Suchverfahren

Die Anwendung von Konsistenztechniken führt i. A. nicht zu einer Lösung eines CSP. Es ist daher notwendig, zusätzliche Algorithmen zum Auffinden von Lösungen anzuwenden. Diese Suchverfahren können nach dem Herstellen einer lokalen Konsistenz, oder kombiniert mit Konsistenztechniken zum Einsatz kommen (vgl. Güsgen 2000, S. 275).

Suchverfahren zur Bestimmung von Lösungen in einem CSP werden unterschieden in systematische und stochastische Verfahren. In diesem Abschnitt werden ausschließlich systematische Suchverfahren vorgestellt. Systematische Verfahren werden auch *globale Suchverfahren* genannt, da sie auf das CSP in seiner gesamten Struktur angewendet werden. Im Gegensatz zu stochastischen Verfahren sind sie vollständig, d. h. es ist garantiert, dass alle existierenden Lösungen gefunden werden, wenn es welche gibt. Stochastische Suchverfahren sind unvollständig und betrachten das CSP anhand lokaler Kriterien. Sie garantieren nicht, dass eine gefundene Lösung globale Gültigkeit besitzt und werden deshalb auch *lokale Suchverfahren* genannt (vgl. Abschnitt 5.2.2, S. 86).

Bei der Anwendung von Suchverfahren nehmen die Constraints i. A. eine passive Rolle ein, da die Strategie zur Lösung des Problems durch den Lösungsalgorithmus vorgegeben

[38]Insbesondere für ein inkrementelles Vorgehen sind Konsistenzverfahren, die die Struktur des Constraint-Netzes modifizieren, weniger geeignet.

allgemeine Suchstrategien	Generate & Test (GT) chronologisches Backtracking (BT)
look-back-Strategien	Backjumping (BJ) Backchecking (BC) Backmarking (BM)
look-ahead-Strategien	Forward Checking (FC) Partial Look-Ahead (PLA) Full Look-Ahead (FLA) bzw. Maintaining Arc Consistency (MAC)

Tabelle 5.2: Klassifizierung von Suchstrategien

ist. Die Constraints werden durch das Suchverfahren dazu genutzt, durch Testen von generierten (Teil-)Lösungen Inkonsistenzen, d. h. ungültige Variablenbelegungen, zu entdecken und ungültige Werte auszuschließen. Bei dieser *a posteriori* Reduktion des Suchraums können die Eigenschaften eines CSP genutzt werden, um effizient zu einer Lösung zu gelangen.

Die Performanz der Suche kann gesteigert werden, indem der Suchraum reduziert wird. Dies kann durch die Kontrollstrategie des Suchalgorithmus beeinflusst werden, durch Herstellen eines lokalen Konsistenzgrades, durch die Reihenfolge, in der die Variablen ausgewählt werden und, wenn nur eine einzige Lösung benötigt wird, durch die Reihenfolge der Werte in den Domänen, mit denen die Variablen während des Suchvorgangs belegt werden (vgl. Dechter und Frost 2002, S. 155 f.). Infolgedessen wurden zwei Arten von Vorgehensweisen entwickelt: Zum einen kann ein Preprozessing des CSP vor der eigentlichen Suche stattfinden. Hier kommen häufig Algorithmen zur Herstellung von Kanten- und Pfadkonsistenz sowie Heuristiken zur Generierung einer statischen Variablenordnung zum Einsatz. Zum anderen gibt es Verfahren, die während der Suche mittels *look-ahead-* und *look-back*-Strategien die Effizienz der Suche beeinflussen. Dies sind z. T. kombinierte Verfahren, die zur Problemreduktion dynamisch während des Suchvorgangs Konsistenztechniken und Heuristiken einsetzen. Die Strategien der hier beschriebenen Suchverfahren können, wie in Tabelle 5.2 dargestellt, in drei Kategorien klassifiziert werden (vgl. Dechter und Frost 1998, S. 3 f.; Dechter und Frost 2002, S. 155 f.; Tsang 1993, S. 119 f.):

1. **Allgemeine Suchstrategien** sind für generelle Problemstellungen geeignet. Sie ziehen keinen Nutzen aus den Constraints bzw. der Struktur eines CSP, um die Effizienz der Suche zu erhöhen.

2. **Look-Back-Strategien** können angewendet werden, wenn Backtracking aufgrund einer inkonsistenten Variablenbelegung notwendig wird. Das Ziel ist es, Zweige im Lösungsbaum, die zu keiner Lösung führen, nicht mehrmals zu durchlaufen. Dies kann auf zwei Arten erreicht werden: Zum einen kann die Inkonsistenz analysiert werden, um zu ermitteln, wie weit die Suche durch das Backtracking zurückgesetzt werden muss, d. h. wie weit im Suchbaum zurückgesprungen werden muss, um irrelevante Backtracking-Punkte zu vermeiden (sog. *backjumping*). Die Suche kann dadurch ggf. direkt an einem Punkt fortgesetzt werden, der vormals eine Inkonsistenz

ausgelöst hat. Zum anderen können durch *look-back*-Strategien Inkonsistenzen bei der Belegung von Constraint-Variablen erfasst werden. Sollte Backtracking notwendig werden ist auf diese Weise sichergestellt, dass inkonsistente Variablenbelegungen nicht wiederholt getestet werden.

3. **Look-Ahead-Strategien** besitzen eine vorausschauende Komponente. Sie werden von kombinierten Verfahren angewendet, die jeweils nach der Festlegung eines Wertes durch den Suchalgorithmus Konsistenztechniken und Constraint-Propagation nutzen, um eine Reduzierung des Suchraums zu erzielen und die Effizienz der Suche zu steigern. Anschließend, nach dem Durchführen der Propagation und den damit einhergehenden Einschränkungen der Wertebereiche der Constraint-Variablen, kann mittels Heuristiken bestimmt werden, welche Variable als nächstes mit welchem Wert aus ihrer Domäne belegt werden sollte, um möglichst schnell zu einer Lösung zu gelangen (vgl. Abschnitt 5.2.5, S. 128).

Im Folgenden werden eine Reihe von systematischen Suchverfahren vorgestellt, die zur Bestimmung von Lösungen in FCSPs geeignet sind. Die Auflistung erhebt keinen Anspruch auf Vollständigkeit, sondern gibt einen Überblick über die grundlegenden bzw. gängigen Techniken.

5.2.4.1 Generate & Test

Die Strategie des *Generierens und Testens* ist das einfachste, systematische Verfahren. Es wird keinerlei Nutzen aus den Constraints bzw. der Struktur des Constraint-Netzes gezogen. Durch dieses Verfahren werden nach und nach Lösungsvorschläge generiert, d. h. es werden alle Variablen mit einem Wert aus ihrem Wertebereich belegt. Die Auswahl der Werte kann dabei deterministisch oder nichtdeterministisch erfolgen. Diese Lösungsvorschläge werden anschließend mit Hilfe der Constraints auf Inkonsistenzen überprüft, d. h. es wird getestet, ob jedes Constraint erfüllt ist. Kann durch den aktuellen Lösungsvorschlag ein Constraint nicht erfüllt werden, wird eine neue Belegung bzw. ein neuer Lösungsvorschlag generiert und es erfolgt eine Wiederholung des Vorgangs (vgl. Barták 1999a, b; Kolbe 2000; Meyer 1995).

Da dieses Verfahren die Constraints nicht dazu nutzt, die Suche effizienter zu gestalten, müssen, wenn alle Lösungen gefunden werden sollen, der gesamte Suchraum „getestet" und nacheinander alle möglichen Lösungen systematisch generiert werden. Dies ist in der Praxis i. A. nicht effizient anwendbar, da die Anzahl der möglichen Lösungsvorschläge der Anzahl der Elemente des kartesischen Produkts der Wertebereiche $D_1 \times \cdots \times D_n$ der Constraint-Variablen entspricht (vgl. Güsgen 2000, S. 275). Unabhängig von der Problemstellung besitzt dieses Verfahren ein sehr schlechtes Laufzeitverhalten und ist ein denkbar ungünstiges Lösungsverfahren.[39]

[39] Das Generieren von Werten kommt allerdings durchaus in Kombination mit anderen Lösungsverfahren zum Einsatz, z. B. zur (heuristischen) Generierung von Belegungen bei stochastischen Suchverfahren (vgl. Barták 1999a, S. 557; Barták 1999b, S. 9).

5.2.4.2 Backtracking

Backtracking ist eine weit verbreitete Methode zur Problemlösung,[40] die bereits sehr früh entwickelt und danach wiederholt neu eingeführt wurde. Bitner und Reingold (1975, S. 651) führen Lucas (1891) als eine frühe Quelle der Beschreibung einer solchen Methode zum Durchqueren von Irrgärten und Labyrinthen an. Backtracking kommt derzeit in vielen Varianten zum Lösen von kombinatorischen Problemen zum Einsatz. Es stellt eine Verbesserung gegenüber naivem Generate & Test dar, weil bei diesem Verfahren die Variablen nach und nach belegt werden und Inkonsistenzen frühzeitig erkannt werden können. Es wird eine Variable nach der anderen ausgewählt und mit einem Wert aus ihrem Wertebereich belegt. Anschließend wird jeweils anhand der vorhandenen Constraints überprüft, ob die vorher bereits belegten Variablen mit dem neuen Variablenwert „kompatibel“ zueinander, d. h. konsistent, sind. Wenn der zugewiesene Wert ein Constraint verletzt, wird der nächste Wert aus dem Wertebereich der aktuellen Variable ausgewählt und die Variable entsprechend mit diesem Wert belegt. Dies geschieht solange, bis alle Werte aus dem Wertebereich probiert oder ein kompatibler Wert gefunden wurde, und die nächste Variable belegt werden kann. Gibt es keinen Wert im Wertebereich, der nicht schon getestet wurde, wird eine Ebene zurückgegangen (engl. *backtracking*) und der zuvor belegten Variable ein alternativer Wert aus deren Wertebereich zugewiesen. Die Zurücknahme der jeweils letzten Wertezuweisung wird chronologisches *Backtracking* (BT) genannt.[41]

Backtracking erweitert somit eine konsistente Teillösung nach und nach zu einer vollständigen Lösung, sofern eine Lösung existiert. Werden alle Alternativen durchprobiert, ohne dass eine konsistente Belegung gefunden werden kann, ist das Constraint-Netz inkonsistent und es gibt keine Lösung für das CSP.

Die Suche im Lösungsraum lässt sich als gerichteter Graph, einem *Suchbaum*, darstellen, der von dem Lösungsalgorithmus durchlaufen wird. Jeder Knoten und jedes Blatt dieses Suchbaums repräsentiert die Belegung einer Variablen mit einem Wert. Der Suchbaum besitzt genau so viele Ebenen wie das CSP Variablen. In jeder Ebene wird jeweils eine Variable mit Werten belegt. Eine Lösung ist ein Pfad durch diesen Suchbaum, auf dem alle Variablen mit zueinander konsistenten Werten bzgl. der Constraints belegt sind. In Abbildung 5.13 auf der gegenüberliegenden Seite ist der vollständige Suchbaum zu dem Constraint-Beispiel aus Abschnitt 3.6.3 auf Seite 39 f. dargestellt. Die gefüllten Kreise stellen legale Zustände bzgl. der Belegung der Constraint-Variablen `P_FSB_Rate`, `MB_FSB_Rate` und `S_FSB_Rate` dar. Terminale legale Zustände sind durch ein gefülltes Dreieck gekennzeichnet. Die durch unterbrochene Kanten verbundenen Kästchen stellen illegale Zustände (Inkonsistenzen) dar. Sie entstehen, wenn die Belegung einer Variable mit einem Wert fehlschlägt. Lässt sich aus einem Zustand heraus für einen Nachfolger keine legale Wertebelegung finden, spricht man von einem terminalen illegalen Zustand bzw. einem *dead-end*, gekennzeichnet durch ein gefülltes Kästchen (siehe Abbildung 5.14 auf der gegenüberliegenden Seite).

Selbst ein einfacher Algorithmus, der den Suchbaum aus Abbildung 5.13 auf der gegenüberliegenden Seite von links nach rechts aufspannt, würde sofort eine Lösung ohne

[40] Prolog nutzt z. B. Backtracking zur Beantwortung von Anfragen.
[41] Dieses einfache Suchverfahren entspricht einer Tiefensuche.

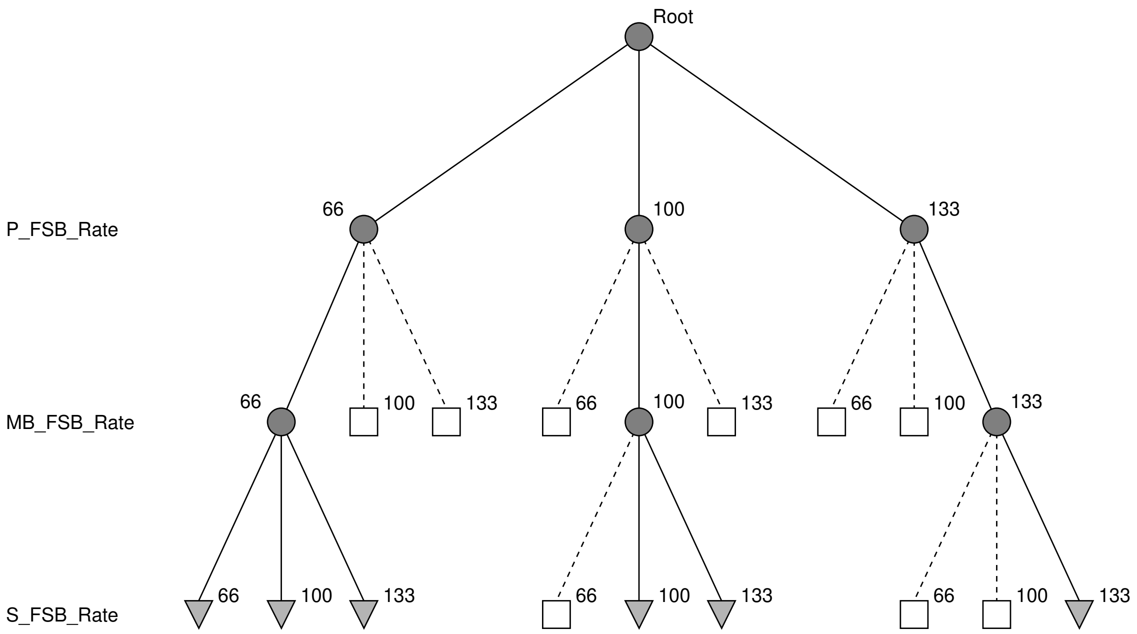


Abbildung 5.13: Beispiel für einen Suchbaum

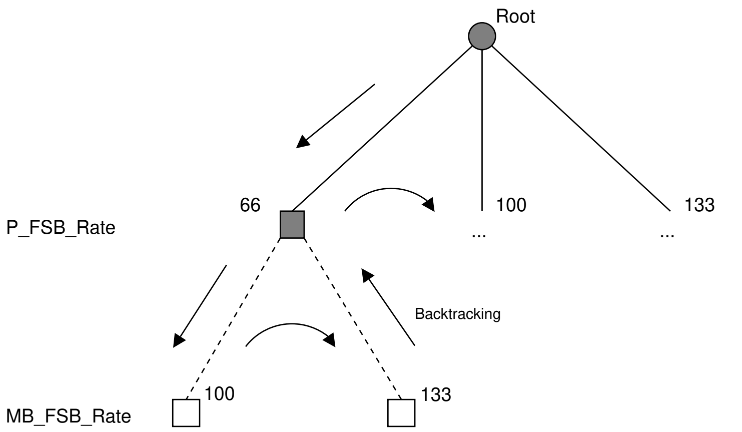


Abbildung 5.14: Beispiel für chronologisches Backtracking

Backtracking generieren können: `P_FSB_Rate = 66`, `MB_FSB_Rate = 66` und `S_FSB_Rate = 66`. Backtracking wird immer dann notwendig, wenn in dem Wertebereich einer Variable kein konsistenter Wert zu den bereits belegten Variablen gefunden werden kann:

Beispiel 5.2.4 *Angenommen, es sind keine Mainboards zugelassen, die einen Front-Side-Bus von 66 MHz zur Verfügung stellen. Der Wertebereich des Parameters* `MB_FSB_Rate` *enthält demnach ausschließlich die Werte 100 und 133. In Abbildung 5.14 auf der vorherigen Seite ist beispielhaft das Backtracking für diesem Fall dargestellt. Die Belegung von* `P_FSB_Rate` *mit 66 ist hier ein terminaler illegaler Zustand, da alle seine Nachfolger Inkonsistenzen bzgl. der vorliegenden Constraints zur Folge hätten. Nachdem der Suchalgorithmus nacheinander die beiden möglichen Werte für* `MB_FSB_Rate` *durchprobiert und Inkonsistenzen zur Belegung von* `P_FSB_Rate` *festgestellt hat, wird der Suchvorgang mit den beiden weiteren Werten für* `P_FSB_Rate` *fortgesetzt.*

In Abbildung 5.15 auf der gegenüberliegenden Seite ist ein einfacher Algorithmus für chronologisches Backtracking von Dechter und Frost (1998, S. 13) dargestellt. Der Algorithmus besteht aus drei Phasen: (1) einer vorwärtsgerichteten Phase, in der die nächste Variable v_{cur} entsprechend der Ordnung ausgewählt wird, (2) einer Phase, in der die aktuelle Teillösung um einen konsistenten Wert für v_{cur} erweitert wird, sofern es diesen gibt und (3) einer rückwärtsgerichteten Phase, in der, falls kein konsistenter Wert für v_{cur} existiert, zur vorherigen Variable zurückgekehrt wird. Mit $\vec{a}_i$ wird eine Teillösung, d. h. das Tupel fortlaufender Werte $(a_1, \ldots, a_i)$ einer partiellen Belegung bezogen auf die gegebene Ordnung der Variablen $v_1, \ldots, v_i$ bezeichnet. Zu beachten ist, dass der Algorithmus genau eine Lösung, die erste, auf die er stößt, zurückgibt. Kann keine Lösung gefunden werden, wird festgestellt, dass das Constraint-Netz inkonsistent ist.

Ein Algorithmus, der mehrere Lösungen bzw. alle Lösungen eines Constraint-Netzes zurückgeben soll, müsste dahingehend erweitert werden, dass er letztendlich den gesamten Suchbaum durchläuft. Die Reihenfolge, in der Lösungen gefunden werden, ergibt sich in Abhängigkeit von der Struktur des Suchbaums und dementsprechend aus der Ordnung der Variablen, anhand der dieser Baum generiert bzw. durchlaufen wird. Soll schnellstmöglich irgendeine Lösung gefunden werden, kann es entscheidend sein, in welcher Reihenfolge die Variablen bzw. die Werte der Variablen ausgewählt werden. Bei einer günstigen Reihenfolge kann u. U. das Testen einer großen Anzahl inkompatibler Belegungen vermieden werden, bevor eine Lösung gefunden wird. Zur Bestimmung der Reihenfolge, in der Variablen mit Werten belegt werden, wurde eine Vielzahl von Heuristiken entwickelt, auf die in Abschnitt 5.2.5 auf Seite 128 ff. eingegangen wird.

Chronologisches Backtracking ist eine vollständige Suche, die systematisch den gesamten Suchraum durchläuft und dabei keinerlei Nutzen aus den Constraints zieht. Dabei ist Backtracking sowohl vollständig (alle Lösungen können gefunden werden) als auch korrekt (alle gefundenen Lösungen erfüllen die Constraints) (vgl. Tsang 1993, S. 120). Das Laufzeitverhalten von Backtracking ist allerdings für die meisten nichttrivialen Probleme exponentiell (vgl. Barták 1999a, S. 557; Tsang 1993, S. 152).

Der Vorteil von Backtracking gegenüber einfachem Generate & Test liegt auf der Hand: Nach jeder Belegung einer Variablen mit einem Wert wird überprüft, ob die Belegung mit

Backtracking
Eingabe: Ein Constraint-Netz C und eine Ordnung auf den Variablen $V = \{v_1, \ldots, v_n\}$.
Ausgabe: Entweder eine Lösung oder das Ergebnis, dass das Netz inkonsistent ist.

0. (Initialisieren.) $cur \leftarrow 0$.
1. (Schritt vorwärts.) Wenn v_{cur} die letzte Variable ist, dann verfügen alle Variablen über eine Wertebelegung; mit dieser Lösung beenden. Ansonsten, $cur \leftarrow cur + 1$. Setze $D'_{cur} \leftarrow D_{cur}$.
2. (Wert auswählen.) Wähle einen Wert $a \in D'_{cur}$, der mit allen bereits instantiierten Variablen konsistent ist. Dabei ist wie folgt vorzugehen:
   (a) Wenn $D'_{cur} = \emptyset$ (v_{cur} ist ein *dead-end*), gehe zu Schritt 3.
   (b) Wähle a aus D'_{cur} und entferne den Wert aus D'_{cur}.
   (c) Teste für jedes auf v_1 bis v_{cur} definierte Constraint, ob es von $\vec{a}_{cur-1}$ und $v_{cur} = a$ verletzt wird. Wenn ja, gehe zu Schritt 2(a).
   (d) Instantiiere $v_{cur} \leftarrow a$ und gehe zu Schritt 1.
3. (Schritt zurück.) Wenn v_{cur} die erste Variable ist, mit dem Ergebnis „inkonsistent“ beenden. Ansonsten, setze $cur \leftarrow cur - 1$. Gehe zu Schritt 2.

Abbildung 5.15: Backtracking-Algorithmus (vgl. Dechter und Frost 1998, S. 13)

den bisherigen Zuweisungen konsistent ist. Erst danach werden weitere Variablen mit Werten belegt. Von Nachteil ist, dass nicht überprüft wird, ob die Wertebelegung evtl. mit später zu belegenden Variablen kollidiert. Dies führt dazu, dass falsch ausgewählte Werte u. U. erst zu einem sehr späten Zeitpunkt entdeckt werden. Abhilfe kann durch *look-ahead*-Strategien geschaffen werden. Ein weiterer Nachteil ist das sog. (engl.) *trashing*, womit der Umstand bezeichnet wird, wenn die Suche ggf. in unterschiedlichen Bereichen des Suchraums wiederholt aus denselben Gründen fehlschlägt (vgl. Mackworth 1977a, S. 100). Dieser und weiterer redundanter Suchaufwand kann mittels *look-back*-Strategien vermieden werden. Nachfolgend werden einige Backtracking-Varianten aufgezeigt, welche die Nachteile von einfachem chronologischem Backtracking reduzieren.

5.2.4.3 Backjumping

Chronologisches Backtracking nimmt u. U. Werte von Variablen zurück, die nicht am aufgetretenen Konflikt beteiligt sind. Wenn z. B. für eine Variable v_i keine konsistente Belegung gefunden werden kann, geht chronologisches Backtracking zurück zu v_{i-1} und versucht für diese Variable eine alternative Wertebelegungen zu finden. Existiert allerdings kein Constraint zwischen v_i und v_{i-1}, wird diese Maßnahme nicht zum gewünschten Erfolg führen: Der Algorithmus geht alle alternativen Werte für v_{i-1} durch und trotzdem liegt bei v_i jedes mal ein illegaler Zustand vor. Erneutes Backtracking und Aufspannen des nachfolgenden Suchbaums wird solange notwendig, bis für diejenige Variable eine alternative Wertebelegung gewählt wird, die tatsächlich für die Inkonsistenz verantwortlich ist.

Backjumping (BJ), eingeführt von *John Gaschnig*, versucht dies zu verhindern, indem in einer Konfliktsituation nicht, wie beim chronologischen Backtracking, nur jeweils ein Schritt zurückgegangen wird. Stattdessen wird versucht zu ermitteln, welche Variablenbelegung „schuld" an dem aufgetretenen Konflikt ist. Bis zu dieser Variable wird „zurückgesprungen" (engl. *to jump back*) und deren Wert direkt geändert (vgl. Gaschnig 1979)[42]. Damit werden u. U. sehr viele, in diesem Fall sinnlose, Backtracking-Schritte vermieden, denn im Normalfall würde beim chronologischen Backtracking zuerst der Wert der letzten Variable verändert werden, dann der vorletzten etc. Der Algorithmus von Gaschnig führt ein Backjumping allerdings ausschließlich in Konfliktsituationen in Blättern des Suchbaums durch. Anderenfalls, d. h. wenn nach dem ersten Rücksprung aufgrund fehlender, konsistenter Werte wiederholt Backtracking in internen Zuständen notwendig ist, wird ausschließlich chronologisches Backtracking angewandt (vgl. Dechter und Frost 2002, S. 158 ff.). Gaschnigs Algorithmus führt Rücksprünge durch, indem mit jeder Variable zusätzlich zur Laufzeit ein Zeiger verwaltet wird, der auf die jeweils letzte Variable verweist, die mit einem Wert der aktuellen Variable inkonsistent war. Sollte Backtracking notwendig werden, kann auf diesen Zeiger zurückgegriffen werden und ein Rücksprung zur den Konflikt auslösenden Variable erfolgen. Die Verwaltung dieser zusätzlichen Zeiger benötigt allerdings bei jedem Konsistenztest etwas vermehrten Berechnungsaufwand (vgl. Dechter 1992, S. 283).

Eine effiziente Variante ist das *graphenbasierte Backjumping* (engl. *graph-based backjumping*, GBJ) von *Rina Dechter*. Beim graphenbasierten Backjumping wird die Ermittlung der den Backtracking-Schritt verursachenden Variablen dahingehend vereinfacht, dass bei einem Konflikt durch Analyse des Constraint-Graphen zu der Variable zurückgesprungen wird, die als letztes die aktuelle Variable durch ein Constraint beschränkt hat (vgl. Dechter 1990a, S. 276 ff.).[43] Im Gegensatz zu anderen Varianten benötigt GBJ daher nur relativ wenig zusätzliche Kapazität zur Ermittlung des Backjumping-Punktes, stellt allerdings nicht sicher, dass wirklich die konfliktauslösende Variable sofort gefunden wird. GBJ nutzt das Wissen über Constraints dazu, bei einem Backtracking-Schritt alle Variablen zu überspringen, die definitiv nicht am aktuellen Konflikt beteiligt sind. Evtl. ist weiteres Backjumping notwendig, um die tatsächlich „schuldige" Variable ausfindig zu machen. Wenn von einer Variable v_i aus ein Rücksprung zur letzten v_i beschränkenden Variable v_{i-k} erfolgt ist, v_{i-k} aber keine Lösung des Konflikts ist, erfolgt wiederum ein Backjumping zur letzten mit einem Wert belegten Variable v_{i-m}, die entweder v_i oder v_{i-k} beschränkt usw.

In Abbildung 5.16 auf der gegenüberliegenden Seite ist beispielhaft (graphenbasiertes) Backjumping anhand des Suchbaums für das Constraint-Netz aus Abbildung 5.7 auf Seite 98 dargestellt:

[42]Zit. nach Bessière und Régin (1996, S. 63); Dechter (1992, S. 283); Dechter (1999, S. 196); Dechter und Frost (2002, S. 158 u. 184); Frost und Dechter (1994, S. 2); Güsgen 2000, S. 279; Prosser (1993a, S. 1); Prosser (1993b, S. 268); Prosser (1995a, S. 185); Tsang (1993, S. 156).

[43]Die Variable, die als letztes mit einem Wert belegt wurde und über ein Constraint mit der aktuellen Variable im Constraint-Graph verbunden ist.

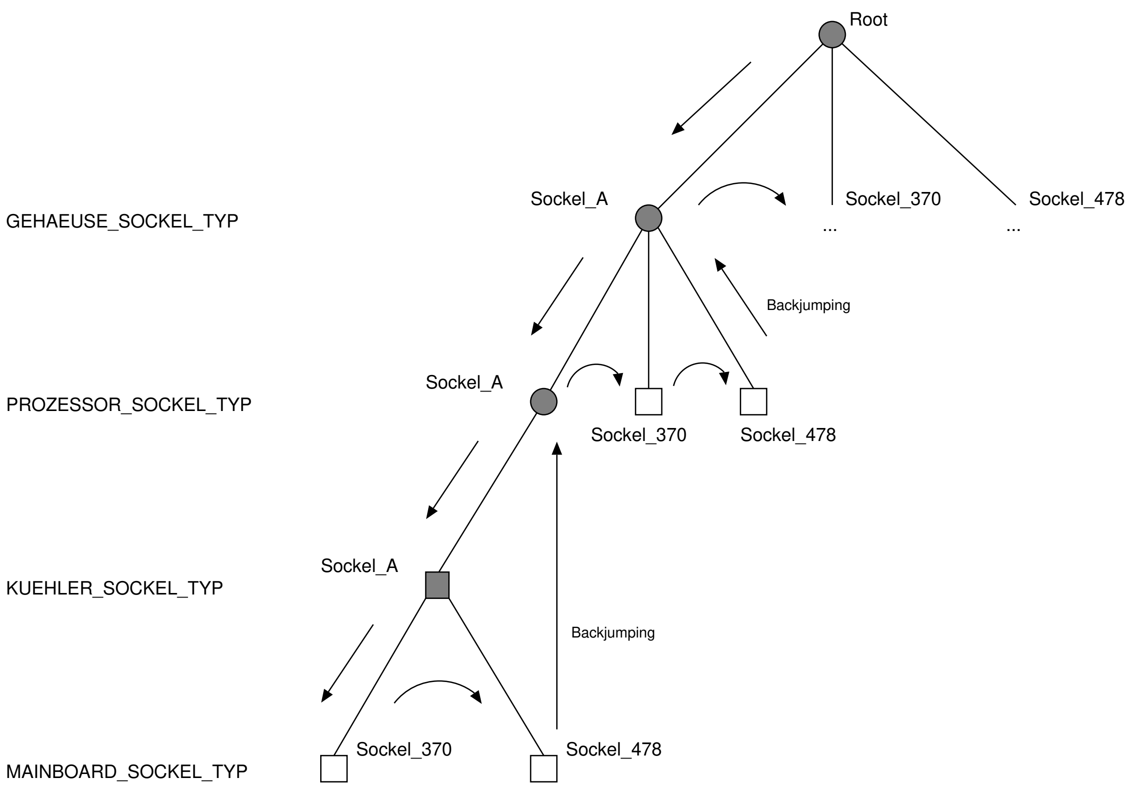


Abbildung 5.16: Beispiel für graphenbasiertes Backjumping

Beispiel 5.2.5 *Angenommen wird, dass die Domänen bereits eingeschränkt wurden, und für die Variable `MAINBOARD_SOCKEL_TYP` nur noch die Werte `Sockel_370` und `Sockel_478` mögliche Belegungen darstellen. Angekommen bei der Belegung dieser Variable, wird durch den Algorithmus ein Konflikt festgestellt, für den keine alternative Belegung ermittelt werden kann. Die letzte durch ein Constraint mit `MAINBOARD_SOCKEL_TYP` verbundenen Variable ist `PROZESSOR_SOCKEL_TYP`, zu der nun unter Umgehung von `KUEHLER_SOCKEL_TYP` zurückgesprungen wird. Da auch für diese Variable keine gültige alternative Belegung ermittelt werden kann, ist nochmals Backjumping erforderlich (aufgrund der Struktur des Constraint-Netzes in diesem Fall äquivalent zu chronologischem Backtracking).*

Eine Zusammenführung der Vorteile von Gaschnigs Backjumping und GBJ stellt *konfliktbasiertes Backjumping* (engl. *conflict-directed backjumping*, CBJ) von *Patrick Prosser* dar (vgl. Prosser 1993b, S. 277 ff.). Der Algorithmus von Prosser baut auf dem Schema von graphenbasiertem Backjumping auf und ist dadurch in der Lage, mehrere Backjumping-Schritte hintereinander auszuführen. Allerdings wird bei Konflikten nicht auf Informationen zurückgegriffen, die sich aus dem Constraint-Graphen ableiten lassen, sondern es werden, wie bei Gaschnigs Backjumping, zusätzliche Informationen bereits während der Suche in Form von sog. *conflict sets* für jede Variable generiert. Aufgrund dieser *conflict sets* kann exakt jeweils die Variable mit der konfliktauslösenden Belegung ausfindig gemacht werden. Immer dann, wenn eine zu testende Belegung der aktuellen Variable v_{cur} in Konflikt mit einer bereits belegten Variable v_i steht, wird v_i dem *conflict set* von v_{cur} hinzugefügt. Gibt es keine weiteren alternativen Werte für die Belegung von v_{cur}, erfolgt ein Backjumping zu der Variable v_i im *conflict set* von v_{cur}, die sich am tiefsten im Suchbaum befindet und damit zuletzt vor v_{cur} mit einem Wert belegt wurde.[44]

Die genannten Backjumping-Varianten werden auch unter dem allgemeinen Begriff *abhängigkeitsgesteuertes Backtracking* zusammengefasst (engl. *dependency-directed backtracking*, DDBT) (vgl. Dechter und Frost 2002, S. 184; Tsang 1993, S. 137). Der Einsatz dieser Strategie ermöglicht es, konfliktauslösende Entscheidungen im Suchbaum zu identifizieren und diese zu korrigieren. Abhängigkeitsgesteuertes Backtracking kann auf vielerlei Problemstellungen angewendet werden.[45] Die Frage nach der Effizienz der Strategie wird beeinflusst von der Problemstellung, die entscheidend dafür ist, ob sich der entstehende Overhead lohnt. Die Analyse von Konflikten anhand von Constraints (GBJ) bzw. *conflict sets* (CBJ) ermöglicht i. A. einen effizienten Einsatz von abhängigkeitsgesteuerten Strategien in CSPs.

5.2.4.4 Backchecking und Backmarking

Backchecking (BC) und *Backmarking* (BM), wie von *Robert M. Haralick* und *Gordon L. Elliot* vorgestellt, sind Erweiterungen, die nach dem *look-back*-Prinzip arbeiten und dazu geeignet sind, die Anzahl von u. U. rechenintensiven Konsistenztests eines Backtracking-Suchverfahrens zu verringern (vgl. Haralick und Elliot 1980, S. 271 ff.). Backchecking erreicht dies, indem sich der Algorithmus Inkompatibilitäten zwischen neu zu belegenden Variablen und bereits mit Werten belegten Variablen merkt. Wenn z. B. eine Variable v_j mit b belegt werden soll und eine Inkompatibilität mit der Belegung a einer vorher bereits belegten Variable v_i festgestellt wird, merkt sich der Algorithmus dies und wird solange nicht mehr versuchen, v_j mit b zu belegen, wie v_i mit a belegt ist. Der Wert b wird dazu (temporär) aus der Domäne von v_j entfernt (vgl. Tsang 1993, S. 147).

Backmarking ist eine Erweiterung von Backchecking, die sich neben den inkompatiblen zusätzlich die bereits getesteten kompatiblen Belegungen merkt, und dadurch die Anzahl der Konsistenztests weiter vermindert. Der Algorithmus merkt sich während des Suchvorgangs in einer einfachen Datenstruktur für jeden möglichen Wert a einer Variablen v_i die früheste vorhergehende Variable v_p in der Belegungsreihenfolge, ab der die Teilbelegung $\vec{a}_p$ in Konflikt mit der Belegung a für v_i steht.[46] Weiterhin merkt sich der Algorithmus für jede Variable v_i die früheste Variable in der Belegungsreihenfolge, deren Wert sich geändert hat, seitdem v_i das letzte Mal mit einem Wert belegt worden ist. Aufgrund dieser Informationen ist der Suchalgorithmus in der Lage zu entscheiden, ob sich die Belegung einer

[44] Damit keine Informationen über bestehende Konflikte verloren gehen, werden bei einem Rücksprung die Variablen des *conflict sets* von v_{cur} zu dem *conflict set* von v_i hinzugefügt (mit Ausnahme von v_i selbst).

[45] Im Zusammenhang mit Prolog wird DDBT auch engl. *intelligent backtracking* genannt. Des Weiteren begründen sich z. B. TMS auf der Anwendung von DDBT (vgl. Dechter und Frost 2002, S. 184; Kumar 1992, S. 38 f.; Russel und Norvig 2002, S. 157).

[46] Constraints mit Variablen, die sich weiter vorne in der Belegungsreihenfolge befinden, müssen zuerst getestet werden.

vorhergehenden Variablen geändert hat, und wenn nicht, ob dessen Belegung weiterhin inkompatibel oder kompatibel mit $v_i = a$ ist.

Backchecking- und Backmarking-Algorithmen wie hier beschrieben, können ausschließlich auf Backtracking-Varianten angewendet werden, die eine statische Reihenfolge bei der Belegung der Variablen $v_1, \ldots, v_n$ nutzen, da sich die Konsistenztests und die Datenstrukturen auf eine fixe Ordnung der Variablen beziehen.

5.2.4.5 Backtracking mit Forward Checking

Das von Haralick und Elliot (1980, S. 270 f.) vorgestellte *Forward Checking* (FC) verwendet eine einfache *look-ahead*-Strategie zur Reduktion des Suchraums und zum Aufspüren von Inkonsistenzen. Forward Checking ergänzt das Backtracking um eine vorausschauende Komponente, die eine eingeschränkte Form von Kantenkonsistenz herstellt.[47] Immer dann, wenn der Backtracking-Suchalgorithmus eine Variable mit einem Wert belegt, werden die Wertebereiche der noch unbelegten, mit der aktuellen Variable über ein Constraint in direkter Abhängigkeit stehenden Variablen, auf inkonsistente Werte überprüft, die mit dem aktuell gewähltem Wert inkompatibel sind. Werte, die nicht zu der gewählten Belegung konsistent sind, werden aus den Wertebereichen der noch nicht belegten Variablen (temporär) entfernt. Es wird dadurch sichergestellt, dass sich in den Wertebereichen von zukünftig zu belegenden Variablen ausschließlich Werte befinden, die mit sämtlichen bisher belegten Variablen kompatibel sind. Durch das Schrumpfen der Wertebereiche wird der Suchraum ggf. drastisch eingeschränkt. Wenn die Domäne einer noch unbelegten Variable keine Werte mehr enthält, kann dadurch u. U. bereits sehr früh festgestellt werden, dass die aktuell gewählte Belegung keine mögliche Lösung darstellt. Stattdessen wird an dieser Stelle ein Backtracking initiiert.

Beispiel 5.2.6 *Ein Beispiel für Forward Checking bezogen auf das Constraint-Beispiel aus Abschnitt 3.6.3 auf Seite 39 f. ist in Tabelle 5.3 auf der nächsten Seite dargestellt. In Zeile 2 können durch die Belegung der Variable `P_FSB_Rate` mit dem Wert 100 aus den Domänen der restlichen Variablen bereits einige Werte entfernt werden, die nicht mehr durch den Suchalgorithmus getestet werden müssen. In Zeile 3 kann direkt der kompatible (und einzig verbliebene) Wert für `MB_FSB_Rate` ausgewählt werden. Der erste Wert für die Variable `S_FSB_Rate`, der in Zeile 4 ausgewählt wird, führt sofort zu einer Lösung des Problems. Backtracking ist nicht erforderlich.*

Forward Checking ist ähnlich dem Backchecking bzgl. dem Entfernen inkonsistenter Werte, die inkompatibel zu einem aktuell gewählten Wert sind, bezieht sich allerdings im Gegensatz zum Backchecking-Algorithmus nicht auf eine Optimierung der rückwärtsgerichteten Backtracking-Phase, sondern versucht im Vorfeld Konflikte zu vermeiden. Wenn v_i mit der Belegung a und v_j mit b inkompatibel zueinander sind, und v_i vor v_j belegt wird, dann wird Forward Checking b in dem Augenblick aus der Domäne von v_j entfernen, wenn v_i mit a belegt wird. Backchecking würde b aus der Domäne von v_j entfernen, in

[47] Forward Checking entspricht der Kombination des Backtracking-Algorithmus mit der `REVISE`-Routine aus Abbildung 5.4 auf Seite 94.

	P_FSB_Rate			MB_FSB_Rate			S_FSB_Rate		
1. initiale Domänen	66	100	133	66	100	133	66	100	133
2. `P_FSB_Rate` = 100	~~66~~	100	~~133~~	~~66~~	100	~~133~~	~~66~~	100	133
3. `MB_FSB_Rate` = 100		100			100			100	133
4. `S_FSB_Rate` = 100		100			100			100	~~133~~
5. Lösung		100			100			100	

Tabelle 5.3: Beispiel für Forward Checking

dem Augenblick wenn v_j belegt werden soll. Obwohl Backchecking eine Reihe von Konsistenztests vermeidet und solche aufschiebt, die u. U. später unnötig sind (die Belegung v_i mit a kann sich bereits wieder geändert haben, wenn v_j an der Reihe ist) ist Forward Checking dem Backchecking überlegen, sowohl was die Anzahl der Backtracking-Schritte (und damit die Größe des Suchbaums) als auch die Anzahl der getesteten Wertebelegungen betrifft (vgl. Haralick und Elliot 1980, S. 271; Tsang 1993, S. 147). Forward Checking erkennt Konflikte aufgrund der *look-ahead*-Strategie früher und ist dadurch in der Lage, ggf. früher Backtracking-Schritte durchzuführen. Obwohl Forward Checking mehr Rechenaufwand als einfaches Backtracking benötigt, sobald eine neue Variable mit einem Wert belegt wird, führt dieser Mehraufwand i. d. R. dazu, dass insgesamt das Problem effizienter gelöst werden kann. Ein grundsätzliches Ablaufschema von *look-ahead*-Algorithmen ist dem Flussdiagramm in Abbildung 5.17 auf Seite 124 zu entnehmen.

Eine Variante ist das von Dent und Mercer (1994a, b) vorgestellte *Minimal Forward Checking* (MFC)[48], welches ggf. nicht jeden Wert von noch unbelegten Variablen auf Konsistenz mit der aktuellen Variable testet. Um festzustellen, dass die aktuelle Variablenbelegung zu einem *domain wipe out* führt, ist es ausreichend, für alle noch unbelegten Variablen lediglich einen einzigen konsistenten Wert pro Variable zu ermitteln.[49] Zusätzliche Konsistenzüberprüfungen werden zu einem späteren Zeitpunkt durchgeführt, sobald diese notwendig werden. Die Anwendung von MFC führt im Vergleich zu normalem FC zu einer signifikanten Verbesserung durch eingesparte Konsistenztests, wenn es auf umfangreiche CSPs angewendet wird, d. h. CSPs mit vielen Variablen und großen Wertebereichen. Weitere aktuelle Erweiterungen des ursprünglichen Forward Checking, ein generelles Schema sowie eine Reihe darauf basierender Algorithmen werden von Bacchus (2000) vorgestellt.

5.2.4.6 Backtracking mit Look-Ahead

In der Literatur wird zwischen der Strategie des *Partial Look-Ahead* (PLA) und der des *Full Look-Ahead* (FLA) unterschieden (vgl. Haralick und Elliot 1980, S. 267). Backtracking-Algorithmen, die während des Suchvorgangs *Full Look-Ahead* einsetzen, werden auch *Maintaining Arc Consistency* (MAC) genannt, unter diesem Namen vorgestellt von Sabin

[48] Auch: *Lazy Forward Checking* (LFC) (vgl. Kwan und Tsang 1996a).

[49] Das Konzept ähnelt der dem Algorithmus `AC-6` von Bessière (1994a) zugrunde liegenden Idee, für die Werte der Variablen vorerst lediglich eine konsistente Wertebelegung pro Constraint zu berechnen (vgl. Abschnitt 5.2.3.3, S. 96). Weitere konsistente Belegungen werden bei Bedarf erzeugt (vgl. Dent und Mercer 1994a, S. 432).

und Freuder (1994a, b).[50] Beide Strategien sind eine Weiterführung des Forward Checking. Der Unterschied besteht darin, dass die in jedem Schritt des Suchalgorithmus hergestellte lokale Konsistenz eine höhere Form von Kantenkonsistenz als beim FC ist. Während ein Suchalgorithmus mit PLA in jedem Schritt gerichtete Kantenkonsistenz herstellt (DAC), wird im Verlauf einer Suche mit FLA bzw. MAC in jedem Schritt ein Algorithmus aufgerufen, der vollständige Kantenkonsistenz (AC) für die bisher noch unbelegten Variablen herstellt.[51]

Der Unterschied von PLA und FLA zu FC besteht demnach in dem Umfang der Propagation der Variablenwerte und einer erhöhten Anzahl von Konsistenztests. Während FC ausschließlich die Konsistenz zwischen der aktuellen Variable und mit dieser über ein Constraint in direkter Verbindung stehender Variablen überprüft, werden durch PLA und FLA auch diejenigen Variablen berücksichtigt, die nicht unmittelbar von der aktuellen Variable eingeschränkt werden. Wenn durch diese vermehrte Propagation eine Domäne keine Werte mehr enthält, bedeutet dies wie beim FC, dass das Problem mit der aktuellen Teilbelegung nicht lösbar ist und ein Backtracking ausgelöst werden muss. Ein allgemeines Ablaufschema von *look-ahead*-Algorithmen ist in Abbildung 5.17 auf der nächsten Seite zu sehen.

Da PLA und FLA bzw. MAC in jedem Schritt der Suche einen höheren Konsistenzgrad als FC herstellt, werden durch diesen Algorithmus mehr inkonsistente Werte herausgefiltert, und der Suchraum dadurch stärker reduziert. Allerdings erhöht sich durch die gestiegene Anzahl Konsistenztests ebenfalls der Berechnungsaufwand dieses Verfahrens, so dass der Aufwand für die Herstellung eines Konsistenzgrades größer werden kann, als der erzielte Gewinn durch die Reduzierung des Suchraums (vgl. Güsgen 2000, S. 279). Dies kann auch darin resultieren, dass die genannten *look-ahead*-Algorithmen bei bestimmten Problemstellungen mehr Aufwand als normales Backtracking ohne *look-ahead* benötigen (vgl. Barták 1999a, S. 559).[52] Es gilt einen Kompromiss zwischen einem möglichst hohem Konsistenzgrad und damit Ausschluss möglichst vieler inkonsistenter Werte, und einem geringen Overhead des Konsistenzalgorithmus zu finden. Generell gilt, wie bereits in Abschnitt 5.2.3.5 auf Seite 103 zu Konsistenzverfahren angesprochen, dass sich das Herstellen eines höheren Konsistenzgrades in Abhängigkeit von dem Problem immer dann lohnt, je stärker sich die Variablen eines CSP gegenseitig beschränken, so dass sich die Domänen

---

[50] Entsprechend dem eingesetzten AC-Algorithmus werden MAC-Algorithmen z. B. mit MAC-3, MAC-4 etc. benannt. Sie sind moderne Varianten des DEEB-Algorithmus (*Domain Element Elimination with Backtracking*) von Gaschnig (1979) (zit. nach Dechter und Meiri 1994, S. 237; Dechter und Frost 1998, S. 45 f.; Gent und Prosser 2000, S. 3; Grant und Smith 1995, S. 2; Prosser 1995b, S. 3 u. 4) bzw. dessen Vorgänger, dem Algorithmus CS2 von Gaschnig (1974) (zit. nach Grant und Smith 1995, S. 2; Prosser 1995b, S. 3; Sabin und Freuder 1994a, S. 12; Sabin und Freuder 1994b, S. 126).

[51] In den ursprünglichen von Haralick und Elliot (1980) vorgestellten Algorithmen wird allerdings im Gegensatz zum MAC-Algorithmus nur eine schwächere Form von AC bzw. DAC erreicht, bedingt dadurch, dass keine eigentliche Propagation im Sinne eines Kantenkonsistenz-Algorithmus vorgenommen wird (vgl. Tsang 1998, S. 355 f.).

[52] Für sehr einfache, unterbestimmte Probleme, die eine große Anzahl Lösungen aufweisen (engl. *underconstrained problems*), zahlt sich das Preprozessing durch *look-ahead*-Mechanismen u. U. nicht aus. Wenn sogar das durch Forward Checking angewendete *look-ahead* zu zeitintensiv und der Gewinn zu gering ist, empfiehlt sich der Einsatz von Backtracking bzw. Backjumping ohne *look-ahead* (vgl. Frost und Dechter 1996a, S. 13).

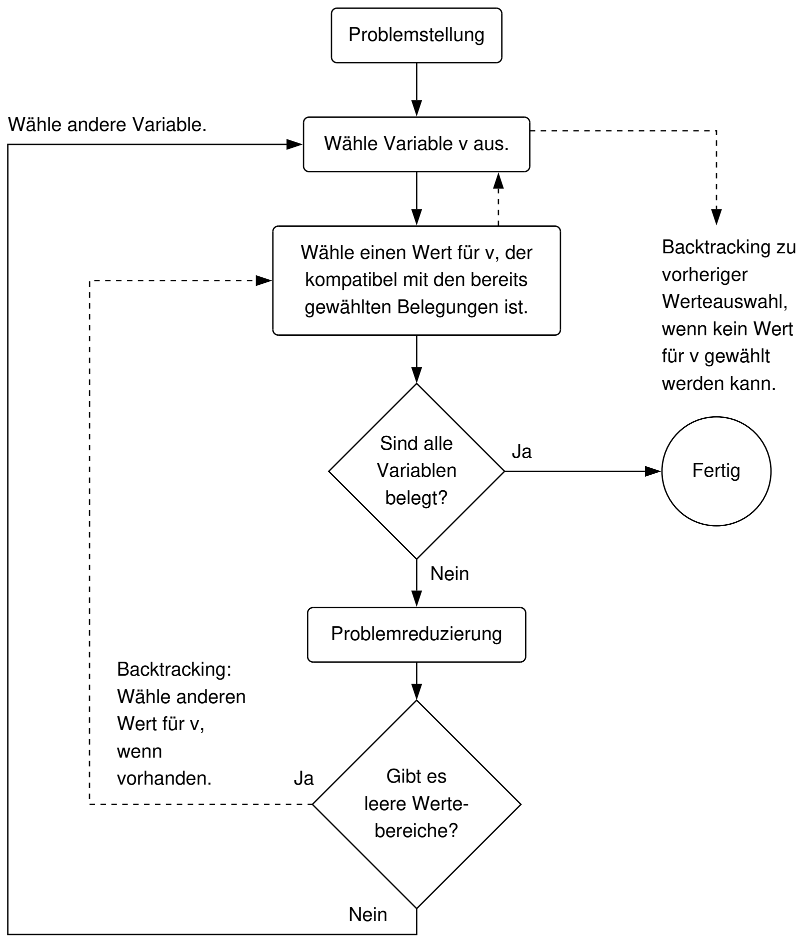


Abbildung 5.17: Ablaufschema von *look-ahead*-Algorithmen (vgl. Tsang 1993, S. 126)

der Variablen durch die angewendeten Konsistenzverfahren um eine möglichst hohe Anzahl Werte reduzieren lassen (vgl. Tsang 1993, S. 136).

Um festzustellen, welcher Konsistenzgrad am besten geeignet ist, einen Suchalgorithmus zu verbessern, ist allerdings eine genauere Betrachtung des konkreten Problems erforderlich. Von Haralick und Elliot (1980) wurde die Performanz verschiedener Suchverfahren zur Lösung von CSPs empirisch untersucht und festgestellt, dass die Verwendung von FC aufgrund der geringeren Anzahl von Konsistenztests zu einem besseren Laufzeitverhalten führt. Die Herstellung von höheren Konsistenzgraden zahlt sich demnach i. A. nicht aus. Lange Zeit haben diese und weitere Untersuchungen dazu geführt, dass *look-ahead*-Mechanismen, die mehr Konsistenztests als FC durchführen, grundsätzlich als weniger effektiv angesehen wurden, und FC sich zum Standard-Suchverfahren für CSPs etablieren konnte. Ursache hierfür war allerdings das fehlende Verständnis für das Verhalten von Suchalgorithmen auf unterschiedlich strukturierten CSPs (vgl. Grant und Smith 1995, S. 4). So wurden empirische Untersuchungen lange Zeit überwiegend auf CSPs mit relativ wenigen Variablen[53] und hoher Constraint-Dichte durchgeführt, wie z. B. dem n-Damen-Problem[54] (vgl. Haralick und Elliot 1980, S. 274 ff.) und dem Zebra-Problem (vgl. Dechter 1990a, S. 308; Prosser 1993b, S. 290 ff.; Smith 1992, S. 37). Ein FC-Suchalgorithmus ist hier Lösungsverfahren mit höherem Konsistenzgrad überlegen, da weniger Konsistenztests vorgenommen werden, und der Overhead durch das Konsistenzverfahren relativ gering ist. FLA- bzw. MAC-Algorithmen sind für derart strukturierte CSPs, bedingt durch die in diesem Fall hohen *look-ahead*-Kosten, weniger gut geeignet (vgl. Grant und Smith 1996, S. 179). Es gibt allerdings eine Vielzahl von Problemen, in denen Verfahren mit mehr *look-ahead* dem einfachen FC überlegen sind. Frost und Dechter (1996a, b) stellen daher einige Varianten vor, die FC und FLA in einem Algorithmus vereinen und beides jeweils nach Bedarf einsetzen.

Mehr Aufwand durch *look-ahead* ist dann besonders sinnvoll, wenn ein stark beschränktes Problem mit niedriger Constraint-Dichte vorliegt, so dass der Overhead des Konsistenzverfahrens aufgewogen wird mit dem Gewinn durch die Reduktion des Suchraums (vgl. Bessière und Régin 1996, S. 7). Ein solches CSP besteht im Idealfall aus relativ wenigen Constraints, in dem allerdings die vermehrte Propagation dazu führt, dass die Domänen der Variablen stark eingeschränkt werden können. Neuere Untersuchungen haben gezeigt, dass insbesondere aktuelle MAC-Algorithmen (vgl. Sabin und Freuder 1997), die einen höheren Konsistenzgrad als FC herstellen, für zufällig generierte CSPs, die stark beschränkt und umfangreicher sind, als die von Haralick und Elliot (1980) untersuchten, die bessere Wahl darstellen (vgl. Bessière und Régin 1996; Frost und Dechter 1996a; Gent und Prosser 2000; Grant und Smith 1996; Sabin und Freuder 1994a).[55] Je umfangreicher das zu lösende Problem, umso eher lohnt sich demnach die Herstellung eines höheren Konsistenzgrades zur Problemreduktion und Vereinfachung der Lösungssuche.[56] Insgesamt muss aber festgehalten werden, dass kein Verfahren dem anderen grundsätzlich überlegen ist. Jedes verfügt über Vorteile in bestimmten Bereichen, da ihr Verhalten problemabhängig von der Struktur des CSP ist. Während die Anwendung von FC bei CSPs mit hoher Constraint-Dichte und schwacher Beschränkung zu empfehlen ist, sollten MAC-Algorithmen eher bei CSPs mit geringerer Constraint-Dichte und stärkerer Beschränkung eingesetzt werden (vgl. Gent und Prosser 2000, S. 9 ff.).

5.2.4.7 Hybride Ansätze

Unter einem hybriden Suchverfahren wird die Kombination von *look-back*- und *look-ahead*-Strategien innerhalb eines Suchverfahrens verstanden. In wissenschaftlichen Veröffentlichungen (vgl. Bessière und Régin 1996; Chen und van Beek 2001; Grant und Smith 1996;

[53] Bis 1993 wurden empirische Untersuchungen lediglich auf relativ begrenzte Probleme mit bis zu 25 Variablen angewendet. Die vorherrschende Meinung war, dass ausschließlich Verfahren mit einem geringen Overhead effektiv anzuwenden sind (vgl. Dechter und Frost 1998, S. 52).

[54] Jede Variable wird durch alle anderen Variablen beschränkt.

[55] Die empirischen Untersuchungen von Haralick und Elliot (1980) beschränken sich auf das n-Damen-Problem mit $4 \leq n \leq 10$.

[56] Untersuchungen lassen darauf schließen, dass für sehr umfangreiche und stark beschränkte Probleme auch pfadkonsistenz-basierte *look-ahead*-Verfahren effektiv eingesetzt werden können (vgl. Dechter und Frost 2002, S. 178).

Haralick und Elliot 1980; Prosser 1993a, b, 1995a, b) gibt es eine lang anhaltende Diskussion darüber, welche Kombination am besten geeignet ist, einfaches, chronologisches Backtracking zu erweitern und damit effizienter zu gestalten. Es sind eine ganze Reihe von Mischformen denkbar, in denen komplementäre *look-back-* und *look-ahead*-Erweiterungen in einem Algorithmus zusammengeführt werden können. Aber auch untereinander lassen sich z. B. *look-back* Verfahren kombinieren. So kann Backjumping relativ einfach um Backmarking zu einem Algorithmus BMJ bzw. BM-CBJ erweitert werden, so dass durch BJ bzw. CBJ *trashing* verhindert wird (insgesamt dadurch weniger Knoten im Constraint-Graphen durchlaufen werden müssen) und zusätzlich, bei den verbleibenden Knoten, die redundanten Konsistenztests durch BM vermieden werden (vgl. Prosser 1993b, S. 281 ff.). Ferner lassen sich die bisher vorgestellten *look-ahead*-Strategien mit unterschiedlichen Backjumping-Varianten kombinieren. Allerdings verhalten sich *look-back-* und *look-ahead*-Strategien nicht vollständig orthogonal zueinander, sondern beeinflussen sich in ihrer Wirkungsweise gegenseitig. Je höher der während eines Suchvorgangs hergestellte Konsistenzgrad ist, umso weniger wirksam werden *look-back*-Strategien und umso mehr fällt deren in diesem Fall nutzloser Overhead ins Gewicht (vgl. Chen und van Beek 2001, S. 62).

Prosser (1993a, 1995a) stellt einen Algorithmus FC-BM vor, der Forward Checking mit Backmarking-Mechanismen kombiniert, wodurch sich der Suchaufwand weiter reduziert. Effizienter wirkt es sich allerdings aus, wenn anstatt eines chronologischen Backtrackers eine Backjumping-Variante zum Einsatz kommt. In der Tat bietet es sich an, Forward Checking zur Vermeidung von *trashing* grundsätzlich mit einem Backjumping-Verfahren zu kombinieren. Entsprechend wurde von Prosser (1993b, S. 289 f.) Forward Checking mit (konfliktbasiertem) Backjumping in dem Algorithmus FC-CBJ vereint. FC-CBJ kann ebenfalls wiederum um Backmarking erweitert werden (FC-BM-CBJ), was sich allerdings nach der empirischen Untersuchung von Prosser (1995a) weniger stark effizienzsteigernd auswirkt, und in seltenen Fällen, durch wechselseitige Beeinflussung der Verfahren, sogar zu einer Verschlechterung der Performanz führen kann.[57]

Aufgrund der empirisch ermittelten Ergebnisse von Prosser (1993b, S. 290 ff.) galt FC-CBJ anderen Kombinationen lange grundsätzlich als überlegen. Auch bei Güsgen (2000, S. 279 f.) findet sich die Aussage, dass die Herstellung eines höheren Konsistenzgrades für *look-ahead*-Verfahren aufgrund des erhöhten Aufwands i. A. nicht lohnt. Von Prosser (1995b) wird allerdings bereits in Anlehnung an Sabin und Freuder (1994a, b) dargelegt, dass mehr Constraint-Propagation durch hybride Verfahren, die während des Suchvorgangs höhere Grade lokaler Konsistenz herstellen, in Abhängigkeit von der Problemstellung zu einer effizienteren Suche führen kann. Um den von Sabin und Freuder (1994a) vorgestellten MAC-Algorithmus (der ebenso wie Forward Checking chronologisches Backtracking einsetzt) weiter zu verbessern, wird MAC von Prosser (1995b) daher um konfliktbasiertes Backjumping erweitert: MAC-CBJ. Wie von Grant und Smith (1996) dargelegt, profitiert MAC allerdings weniger als FC von einer *look-back*-Strategie. Da MAC den Lösungsraum

[57]Neben Forward Checking wurde später auch Minimal Forward Checking, welches bereits in der „Grundversion“ bei leicht erhöhtem Overhead in der Lage ist, ggf. eine beträchtliche Anzahl Konsistenztests einzusparen, um Backmarking und konfliktbasiertes Backjumping erweitert: MFC-BM2-CBJ (vgl. Kwan und Tsang 1996b bzw. ausführlicher Kwan und Tsang 1996a).

stärker reduziert als FC, wird eine Backtracking-freie Suche über größere Regionen erlaubt, wodurch Backjumping seltener zum Einsatz kommt. Die Kombination von MAC mit CBJ hat demnach auf die meisten Suchvorgänge weniger Einfluss, garantiert allerdings eine stabile Performanz bei fast allen Problemstellungen.

Unter den Veröffentlichungen finden sich eine Reihe weiterer, z. T. kontroverser Studien zu dem Verhalten der Kombination MAC-CBJ, die sich an der Diskussion beteiligen, wie viel *look-ahead* letztendlich sinnvoll ist, bzw. ob bei dem entsprechendem *look-ahead* von MAC eine *look-back*-Strategie noch sinnvoll einsetzbar ist.[58] So kann nach Bessière und Régin (1996, S. 72) die Erweiterung von MAC um CBJ nicht einfach als eine Verbesserung von MAC angesehen werden, wie CBJ für BT und FC-CBJ für FC. Die Ergebnisse der empirischen Untersuchung von Bessière und Régin (1996, S. 66 ff.) lassen annehmen, dass CBJ die Suche mittels MAC verlangsamt, da eine signifikante Anzahl Konsistenztests eingespart werden müsste, um den Overhead von CBJ aufzuwiegen. Die durch CBJ eingesparten Backtracking-Schritte und Konsistenztests reichen demnach allerdings i. A. nicht aus, diesen zusätzlichen Aufwand wieder wettzumachen. Stattdessen wird von Bessière und Régin (1996) empirisch gezeigt, dass durch Ausnutzung einer effizienten Reihenfolge der zu belegenden Variablen CBJ redundant wird (vgl. Abschnitt 5.2.5.1, S. 128). Tatsächlich wird von Chen und van Beek (2001, S. 58 ff.) nachgewiesen, dass eine Variablenordnung existiert, so dass BT niemals mehr Knoten im Constraint-Graphen durchläuft als CBJ.[59] Allerdings wird der Aussage von Bessière und Régin (1996, S. 72 f.) widersprochen, dass CBJ bei Verfahren, die während des Suchvorgangs Kantenkonsistenz herstellen, grundsätzlich sinnlos ist. So wird von Chen und van Beek (2001) neben dem theoretischen Nachweis, dass bei erhöhtem Konsistenzgrad während einer Backtracking-Suche grundsätzlich der Nutzen von Backjumping verringert wird, anhand einer Reihe von empirischen Untersuchungen auf Standardproblemen gezeigt, dass die Ergänzung von MAC um CBJ in der Lage ist, die Effizienz des Suchverfahrens um mehrere Größenordnungen zu verbessern.[60]

Abschließend muss festgestellt werden, dass es nicht *das beste* Lösungsverfahren bzw. eine optimale Kombination gibt, da es eben nicht *das typische* CSP gibt. Um herauszufinden, welches Lösungsverfahren sich für ein bestimmtes Problem besonders eignet, sind eine eingehende Evaluierung der bestehenden Untersuchungsergebnisse zu diesem Thema, die Analyse des konkreten Problems bzw. des Constraint-Netzes sowie u. U. eigene empirische Untersuchungen erforderlich. In Bezug auf ein Constraint-System, welches für unterschiedliche Problemstellungen eingesetzt werden soll, bedeutet dies, das unterschiedliche Lösungsalgorithmen angeboten werden sollten, die je nach Problemstellung flexibel eingesetzt werden können. Eine wichtige Rolle bzgl. der Effizienz eines Lösungsverfahrens spielt, wie bereits angedeutet, die Reihenfolge, in der Variablen mit Werten belegt werden

[58] „Lookahead to the future in order not to worry about the past" (Haralick und Elliot 1980, S. 263).

[59] Diese Reihenfolge ist allerdings von vornherein nicht bekannt, zudem haben Variablenordnungsheuristiken i. A. nicht die Aufgabe CBJ zu simulieren.

[60] In diesem Fall kommt eine Variante GAC-CBJ zum Einsatz, die für das *look-ahead* Kantenkonsistenz für n-äre Constraints herstellt (vgl. Abschnitt 5.2.6.1, S. 136 ff.) und eine modifizierte Implementierung von CBJ verwendet, deren Overhead zur Verwaltung der *conflict sets* implementierungsabhängig verringert wurde.

und damit einhergehend die entsprechenden Heuristiken, die diese Reihenfolge generieren. Der nachfolgende Abschnitt befasst sich daher mit Variablen- und Werteordnungsheuristiken.

5.2.5 Heuristiken zur Variablen- und Werteauswahl

Variablen- und Werteordnungsheuristiken betreffen die Reihenfolge, in der Variablen während eines Suchvorgangs belegt werden und die Reihenfolge, in der die Werte aus den Domänen jeder Variable zugewiesen werden. Diese jeweilige Reihenfolge bzw. Ordnung ist durch die unterschiedliche Generierung des Suchbaums in der Lage, die Effizienz eines Suchverfahrens hinsichtlich der Anzahl der Backtracking-Schritte und der Größe der reduzierten Bereiche des Lösungsraums (bei *look-ahead*-Verfahren) signifikant zu beeinflussen (vgl. Tsang 1993, S. 157). Da sich Variablen- und Werteordnungsheuristiken orthogonal zueinander verhalten, können sie gleichzeitig eingesetzt werden (vgl. Güsgen 2000, S. 276).

5.2.5.1 Variablenordnungsheuristiken

Die Belegung von Variablen mit Werten sollte in einer Reihenfolge vorgenommen werden, so dass eine vollständige, konsistente Belegung mit möglichst wenig Backtracking hergestellt werden kann. Um zu verhindern, dass große Teile von bereits belegten Variablen aufgrund von erst spät festgestellten Inkonsistenzen wiederholt neu belegt werden müssen (*trashing*), ist es sinnvoll „kritische Variablen", von denen viele andere Belegungen abhängig sind, zu Beginn einer Suche zu belegen. Anstatt uninformiert eine Variable nach der anderen zu belegen, werden Heuristiken eingesetzt, um basierend auf der Anzahl der Elemente in den Wertebereichen und den involvierten Constraints festzustellen, wie kritisch eine Variable im Vergleich zu anderen ist (engl. *most constrained first*). Aufgrund dieser Heuristiken wird eine Belegungsreihenfolge generiert. Der Vorgang kann sowohl *statisch*, als Preprozessing vor Beginn einer Suche (engl. *static variable ordering*, SVO), als auch *dynamisch*, während einer Suche, jeweils bevor eine neue Variablenbelegung durchgeführt wird, erfolgen (engl. *dynamic variable ordering*, DVO). Im Gegensatz zu statischen Variablenordnungen variiert bei dynamischen die Belegungsreihenfolge der Variablen in unterschiedlichen Zweigen des Suchbaums (vgl. Barták 2003; Ruttkay 1998).

Da die anfangs generierte Ordnung bei Anwendung von statischen Heuristiken während der gesamten Suche beibehalten wird, ist der Berechnungsaufwand i. A. geringer, als die Anwendung von dynamischen Ordnungsheuristiken. Zudem ist die Anwendung von DVOs nicht in jedem Fall sinnvoll. So ergeben sich z. B. während der Suche mit einem einfachen, chronologischen Backtracking-Algorithmus keine zusätzlich nutzbaren Informationen. Das Constraint-Netz bzw. die Wertebereiche der Constraint-Variablen werden in diesem Fall keinerlei Änderungen unterworfen, die die Reihenfolge einer initialen Variablenordnung beeinflussen würden (vgl. Barták 2003). Anders verhält es sich bei der Anwendung von FC und anderen *look-ahead*-Suchverfahren. Durch die Reduzierung der Wertebereiche der Variablen können sich Änderungen ergeben, deren Ausnutzung durch eine DVO zu einer deutlichen Verbesserung der Effizienz des Suchverfahrens führt. Im Folgenden werden die geläufigsten Variablenordnungsheuristiken vorgestellt:

fail first (FF): Das von Haralick und Elliot (1980, S. 301 ff.) vorgestellte *fail-first*-Prinzip geht zurück auf eine bereits von Bitner und Reingold (1975) vorgeschlagene *search rearrangement*-Methode zur Optimierung von Backtracking-Algorithmen. FF wird häufig mit großem Erfolg eingesetzt[61] und ist außer unter der Bezeichnung *search rearrangement* (vgl. Bitner und Reingold 1975; Kumar 1992; Purdom 1983) auch unter den Namen *minimum remaining values* (MRV) (vgl. Bacchus und Grove 1995; Bacchus und van Run 1995; Russel und Norvig 2002), *minimum domain* (`dom`) (vgl. Bessière und Régin 1996; Chen und van Beek 2001), *dynamic search rearrangement* (DSR) (vgl. Dechter und Meiri 1994; Güsgen 2000) und unter der allgemeinen Bezeichnung DVO (vgl. Frost und Dechter 1994) bekannt. Die Heuristik erstellt eine Belegungsreihenfolge aufgrund des Prinzips, dass Bereiche des Suchbaums, in denen ein Fehlschlagen wahrscheinlicher ist als in anderen, zuerst durchsucht werden sollten.[62] Auf diese Weise soll sichergestellt werden, dass Fehlschläge in einer möglichst frühen Phase der Suche erkannt werden.[63] Infolgedessen wird bei Anwendung der FF-Heuristik versucht, von den verbleibenden Variablen immer diejenige mit dem kleinsten Wertebereich als nächstes mit einem konsistenten Wert zu belegen. Trotz seiner Einfachheit ist FF damit sehr effektiv, denn durch die (aufgrund der geringeren Anzahl zu testender Werte) verminderte Verzweigungstiefe im Suchbaum, ist die Wahrscheinlichkeit tatsächlich höher, mögliche Inkonsistenzen früher festzustellen (vgl. Haralick und Elliot 1980, S. 308). In Kombination mit einfachem BT kann die FF-Heuristik statisch eingesetzt werden, d. h. es wird vor Beginn der Suche eine fixe Variablenordnung aufsteigend bzgl. der Wertebereiche generiert. Zusammen mit *look-ahead*-Suchalgorithmen, während deren Ausführung inkonsistente Werte aus den Domänen von noch unbelegten Variablen entfernt werden, wird FF dynamisch angewendet.

minimal width ordering (MWO): Die von Freuder (1982, S. 27 ff.) beschriebene *minimal-width-ordering*-Heuristik wird auch kurz mit *min-width* (vgl. Frost und Dechter 1994) und `minw` (vgl. Bessière und Régin 1996) benannt. Mit ihrer Hilfe wird versucht, eine Reduzierung des Backtracking vorzunehmen, indem Variablen, die mehr beschränkt werden als andere, in der Reihenfolge der zu belegenden Variablen weiter vorne angeordnet werden. Umgekehrt sollten Variablen, die auf weniger andere Variablen angewiesen sind, später mit einem Wert belegt werden, da für diese Variablen einfacher eine gültige Belegung gefunden werden kann. Die MWO-Heuristik wertet zur Generierung der Reihenfolge allerdings nicht wie FF die Größe der Wertebereiche aus, sondern die Beschränkung der Variablen durch die Constraints. Dazu wird eine Ordnung mit einer möglichst geringen „Weite“ (engl. *width*) erzeugt. Die Weite einer Variable ergibt sich dabei aus der Anzahl der Variablen, die sich in einer gegebenen totalen Ordnung *vor* dieser Variable befinden und im Constraint-

[61] Beispielsweise in in der CLP-Sprache CHIP (vgl. Tsang 1993, S. 178 u. 188).

[62] „To succeed, try first where you are most likely to fail“ (Haralick und Elliot 1980, S. 263).

[63] Ähnlich dem FC, PLA, FLA und anderen *look-ahead*-Verfahren, die ebenfalls dem frühzeitigen Auffinden von inkonsistenten Wertebelegungen dienen.

Graphen *benachbart* sind.[64] Die Weite einer Ordnung ist die maximale Weite aller Variablen unter dieser Ordnung, und die Weite eines Constraint-Netzes ist die minimale Weite aller möglichen Ordnungen.

Beispiel 5.2.7 *In Abbildung 5.18 ist dazu ein Beispiel aufgeführt. Für den in 5.18(a) dargestellten Constraint-Graphen ist in 5.18(b) eine Variablenordnung der Weite 2 angeben (maximale Weite der beteiligten Variablen). Dies ist in diesem Fall zugleich die minimale Ordnung des Constraint-Graphen.*

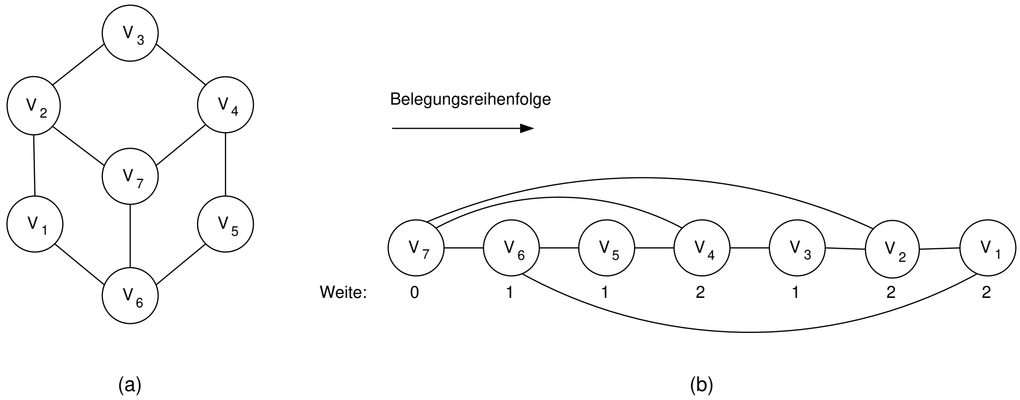


Abbildung 5.18: Die Weite einer Variablenordnung (vgl. Freuder 1982, S. 29)

Wenn es gelingt, diese Ordnung mit minimaler Weite zu finden, ist sichergestellt, dass sich Variablen, auf die viele andere Variablen durch Constraint-Verbindungen angewiesen sind, weiter vorne in der Ordnung befinden, wodurch wiederum die Notwendigkeit von Backtracking verringert wird.[65] Die MWO-Heuristik ist eine statische Heuristik, die vor Beginn einer Suche ausgeführt wird.

maximum cardinality ordering (MCO): Das *maximum-cardinality-ordering* kann als eine Approximation an die Ergebnisse der MWO-Heuristik angesehen werden (vgl. Tsang 1993, S. 179 ff.). Die MCO-Heuristik wird auch unter der Bezeichnung CARD (vgl. Dechter und Meiri 1994) bzw. `card` (vgl. Bessière und Régin 1996) geführt. MCO ist ebenfalls eine statische Variablenordnungsheuristik und hat ebenso wie MWO das Ziel, Backtracking dadurch zu reduzieren, dass Variablen mit vielen Abhängigkeiten möglichst früh mit Werten belegt werden. Die Ordnung wird dazu in umgekehrter Reihenfolge erstellt. Begonnen wird mit einer beliebigen Variable, die das letzte Element in der Belegungsreihenfolge darstellt. Danach wird der Reihe nach

[64] Das heißt es existiert ein Constraint zwischen diesen beiden Variablen.

[65] Freuder (1982, S. 27) weist zudem nach, dass, für ein CSP mit einem strengen Konsistenzgrad > der Weite, eine Lösung gänzlich ohne Backtracking gefunden werden kann – vorausgesetzt die Variablen werden in der entsprechenden Reihenfolge mit Werten belegt. Dies wirkt sich z. B. bei der Behandlung von baumartigen Constraint-Graphen (Weite = 1 für jede beliebige Belegungsreihenfolge) mittels AC bzw. DAC aus (vgl. Abschnitt 5.2.3.5, S. 105).

jeweils die Variable ausgewählt, die mit den meisten bisher ausgewählten Variablen über ein Constraint verbunden ist. Die Weite einer mit der MCO-Heuristik generierten Ordnung ist häufig größer als die einer durch die MWO-Heuristik generierten minimalen Ordnung, kann dafür aber mit weniger Aufwand hergestellt werden.

maximum degree ordering (MDO): Eine der MWO ebenfalls ähnliche aber wesentlich simplere Methode ist die *maximum-degree-ordering*-Heuristik (vgl. Tsang 1993, S. 166), auch kurz DEG (vgl. Dechter und Meiri 1994) bzw. `deg` (vgl. Bessière und Régin 1996; Chen und van Beek 2001) genannt. Das Vorgehen besteht darin, die Variablen einfach absteigend nach ihrer Stelligkeit bzw. dem Grad ihres Vorkommens in den Constraints anzuordnen. Das heißt Variablen, die von vielen Constraints beschränkt werden, befinden sich vorn in der Belegungsreihenfolge. Die MDO-Heuristik ist ebenfalls eine statische Ordnungsheuristik, approximiert bei deutlich geringerem Berechnungsaufwand MWO und verringert das Backtracking während des Suchvorgangs.

minimal bandwidth ordering (MBO): Im Gegensatz zu den bereits angesprochenen Heuristiken ist die *minimal-bandwidth-ordering*-Heuristik (vgl. Tsang 1993, S. 166 ff.) nicht dadurch motiviert, die Notwendigkeit von Backtracking zu verringern, sondern die Abstände zwischen sich gegenseitig beschränkenden Variablen innerhalb der Ordnung (Minimierung der „Bandbreite“). Durch die in der Ordnung möglichst dichte Platzierung von im Constraint-Netz benachbarter Variablen ist gewährleistet, dass im Konfliktfall ein chronologischer Backtracking-Algorithmus kürzere Strecken im Suchbaum überwinden muss, und dadurch möglichst schnell auf die konfliktauslösende Variable trifft. Je kleiner die Bandbreite einer Ordnung ist, umso eher kann an die Stellen zurückgegangen werden, die auslösend für Backtracking sind. Während sich die Bandbreite einer Variablen aus dem maximalen Abstand zu seinen Nachbarn bzgl. des Constraint-Netzes innerhalb einer Ordnung ergibt, ist die Bandbreite einer Ordnung die maximale Bandbreite aller existierender Variablen, und die Bandbreite eines Constraint-Netzes wiederum die minimale Bandbreite aller möglichen Ordnungen. Die MBO-Heuristik ist eine statische Ordnungsheuristik, die, als Preprozessing vor Beginn einer Suche, eine totale Ordnung mit minimaler Bandbreite erzeugt.

Die Anwendung von Ordnungsheuristiken geschieht in Abhängigkeit von der Problemstellung und von dem eingesetzten Lösungsverfahren. MWO, MCO, MDO und MBO sind statische Variablenordnungsheuristiken, die eine Ordnung generieren, bevor der Suchvorgang gestartet wird. Sie nutzen die Struktur des Constraint-Graphen, berücksichtigen jedoch nicht die Veränderungen der Wertebereiche während der Suche durch ein *look-ahead*-Suchverfahren. Die FF-Heuristik hingegen kann statisch und dynamisch eingesetzt werden und berücksichtigt die Domänen der Constraint-Variablen, nicht jedoch die Topologie des Constraint-Graphen.

Die Effizienz der Heuristiken ist problemabhängig. Während FF umso erfolgreicher ist, je signifikanter die Größe der Wertebereiche schwankt bzw. wenn ein Konsistenzverfahren als *look-ahead* zum Einsatz kommt,[66] sind MWO, MCO und MDO effizient anwendbar für Probleme, in denen einige Variablen von mehr Constraints beschränkt werden als andere. Im Umkehrschluss sind diese Ordnungsheuristiken nicht sinnvoll einsetzbar, wenn alle Wertebereiche dieselbe Größe aufweisen und kein Konsistenzverfahren eingesetzt wird (FF) bzw. wenn der Constraint-Graph einen einheitlichen Grad oder gar vollständige Vernetzung aufweist (MWO, MCO und MDO). Die MBO-Heuristik ist ausschließlich zur Optimierung von Suchverfahren mit chronologischem Backtracking geeignet. Ein hoher Vernetzungsgrad im Constraint-Netz wirkt sich hier negativ auf das Ergebnis aus, da das Maximum der Vernetzung die minimale Bandbreite des Constraint-Netzes diktiert (vgl. Tsang 1993, S. 182).[67] In der Vergangenheit hat sich häufig die Überlegenheit von dynamischen gegenüber statischen Variablenordnungsheuristiken gezeigt (vgl. Bacchus und van Run 1995, S. 272; Dechter und Meiri 1994, S. 236; Frost und Dechter 1994, S. 304; Purdom 1983, S. 132). Viele Arbeiten zu diesem Thema kommen zu dem Schluss, dass die Anwendung speziell des FF-Prinzips für viele Anwendungen eine sehr günstige Heuristik ist, die sich insbesondere in Kombination mit FC und FC-CBJ als sehr effizient erwiesen hat (vgl. Bacchus und van Run 1995, S. 270; Bacchus und Grove 1995, S. 308).

Die genannten Heuristiken können zur Effizienzsteigerung miteinander kombiniert werden. Da es für ein Constraint-Netz für gewöhnlich jeweils mehrere Ordnungen gibt, die minimale Weite und minimale Bandbreite aufweisen, ist es z. T. möglich, eine Ordnung zu generieren, die gleichzeitig MWO und MBO aufweist, d. h. beide Minima miteinander vereint. Für Constraint-Netze, in denen dies nicht möglich ist, kann eine Annäherung erreicht werden. Die Kombination von MWO oder MBO mit FF kann auf mehrere Arten erfolgen:

1. Die Auswahl der Variablen erfolgt in erster Linie nach der MWO- bzw. MBO-Heuristik. Bei Konflikten, d. h. wenn mehrere Variablen denselben Vernetzungsgrad besitzen, wird gemäß FF aus diesen die Variable mit dem kleineren Wertebereich ausgewählt (vgl. Tsang 1993, S. 184).

2. Umgekehrt kann die Anwendung des FF-Prinzips im Vordergrund stehen, d. h. die Auswahl erfolgt nach Größe der Wertebereiche. Existieren mehrere Variablen, deren Wertebereiche dieselbe Anzahl Elemente aufweisen, wird aus diesen mittels MWO oder MBO die nächste zu belegende Variable ausgewählt (vgl. Tsang 1993, S. 182 ff.). Als sehr effektiv hat sich die Kombination von FF mit der an MWO angenäherten und weniger rechenintensiven MDO-Heuristik erwiesen (vgl. Frost und Dechter 1995, S. 573). Der resultierende Algorithmus wird von Bessière und Régin (1996) auch FC-CBJ-`dom+deg` genannt. Anstatt wie FC-CBJ-`dom` in Fällen, in denen die Wertebereiche von Variablen dieselbe Anzahl Elemente aufweisen, die nächste Variable

[66] Viele stark beschränkende Constraints wirken sich vorteilhaft bzgl. starker Schwankungen bei Anwendung von *look-ahead*-Verfahren aus und damit entsprechend für die Anwendung der FF-Heuristik.

[67] Je mehr Variablen bzw. Constraints Einfluss auf die Konsistenz der Belegung einer Variable haben, umso mehr Fehlerquellen müssen im Konfliktfall durch Backtracking angesteuert werden, d. h. umso größer wird der Abstand voneinander abhängiger Variablen innerhalb der Ordnung und damit die minimale Bandbreite.

zufällig auszuwählen, berücksichtigt dieser Algorithmus als nächstes die Variable mit dem höheren Vernetzungsgrad im Constraint-Netz.

3. Bei beiden erstgenannten Kombinationsmöglichkeiten steht jeweils eine Heuristik besonders im Vordergrund. Während sich jedoch die FF-Heuristik in CSPs mit geringer Constraint-Dichte weniger gut hinsichtlich der Vermeidung von Backtracking verhält, ist der Einsatz von MDO dort im Vergleich günstiger. Dafür wird MDO umso ungünstiger, je mehr Constraints berücksichtigt werden müssen. Die Heuristiken verhalten sich also invers zueinander, was in den genannten Kombinationen aufgrund der Dominanz jeweils einer Heuristik nicht vollständig ausgeglichen werden kann. Eine Kombination von FF und MDO, die beide Vorteile verbindet und die Nachteile weitestgehend kompensiert, wird von Bessière und Régin (1996, S. 69 ff.) vorgestellt: `dom/deg`. Nach dieser Heuristik wird jeweils die Variable als nächstes mit einem Wert belegt, deren Verhältnis von Größe des Wertebereichs zu Vernetzungsgrad am geringsten ist. Die Heuristik wird von Bessière und Régin (1996) zur Optimierung des MAC-Algorithmus eingesetzt, wobei MAC-`dom/deg` eine gleich bleibende Verbesserung hinsichtlich der Vermeidung von Backtracking, bezogen auf die Constraint-Dichte, erreicht und demnach insgesamt der MAC-`dom+deg`-Variante überlegen ist. Selbiges gilt, wenn auch weniger stark ausgeprägt, für FC-CBJ-`dom/deg`.

Ob sich der betriebene Aufwand zur Herstellung einer Variablenordnung lohnt, ist allerdings problem- bzw. domänenabhängig und muss von Fall zu Fall entschieden werden. Da es sich bei den oben genannten um allgemeine Heuristiken handelt, kann es u. U. domänenspezifische Heuristiken geben, die eine wesentlich effizientere Problemlösung ermöglichen.

5.2.5.2 Werteordnungsheuristiken

Nachdem eine Variable zur Belegung ausgewählt wurde, stehen zumeist mehrere Belegungsmöglichkeiten im Wertebereich zur Auswahl. Die Aufgabe von Werteordnungsheuristiken ist es, hiervon einen Wert auszuwählen und somit insgesamt eine Reihenfolge zu generieren, in der die Variable mit Werten belegt wird. Da diese Werteordnung die Reihenfolge vorgibt, in der während des Suchvorgangs in die Äste des Suchbaums verzweigt wird, sind Werteordnungsheuristiken in der Lage, ebenfalls beträchtlich die Effizienz des Suchvorgangs zu beeinflussen

Während Variablenordnungsheuristiken darum bemüht sind, kritische Variablen möglichst frühzeitig mit einem Wert zu belegen (engl. *most critical variables first*), verfolgen Werteordnungsheuristiken, auch *look-ahead value ordering* (LVO) genannt (vgl. Bessière und Régin 1996; Frost und Dechter 1995), das gegenteilige Prinzip: Sie versuchen zur Vermeidung von Backtracking Werte auszuwählen, die möglichst zur einer konsistenten Belegung aller Variablen führen (engl. *most promising values first*) (vgl. Ruttkay 1998, S. 13). Wenn zuerst Zweige im Suchbaum durchsucht werden, deren Wahrscheinlichkeit zu einer konsistenten Lösung zu führen größer ist als die anderer, kann hierdurch eine erste Lösung früher aufgefunden werden. Werteordnungsheuristiken haben allerdings i. A. keinen Einfluss auf die Effizienz eines Suchverfahrens, wenn alle Lösungen für das CSP gesucht werden sollen, bzw. wenn es für das CSP keine Lösung gibt, da in diesen Fällen alle Werte berücksichtigt werden müssen (vgl. Barták 2003; Dechter und Frost 1998, 2002; Russel und Norvig 2002; Tsang 1993).[68]

Ein häufig angewendetes Vorgehen nach dem *least-constraining-value*-Prinzip zur Einschätzung, wie vielversprechend mögliche Werte zur Belegung einer Variable für das Auffinden einer konsistenten Lösung sind, ist die **min-conflicts**-Heuristik (MC). Der Wert, der die wenigsten Konflikte mit den Werten von noch unbelegten Variablen aufweist und damit die meisten zukünftigen Belegungsmöglichkeiten offen hält, ist demnach derjenige, mit dem die aktuelle Variable zuerst belegt werden sollte. Je weniger Konflikte ein Wert mit potentiellen Belegungen von noch unbelegten Variablen aufweist, umso weiter vorn wird er in der Werteordnung eingereiht (vgl. Minton et al. 1992).

Andere Strategien zur Werteauswahl beruhen auf der Annahme, dass ein Teilproblem eher eine konsistente Lösung aufweist, wenn es keine Variablen mit Wertebereichen umfasst, die nur einen oder sehr wenige Werte beinhalten (vgl. Frost und Dechter 1995, S. 574). Die **max-domain-size**-Heuristik (MD) bevorzugt demnach Werte, die den größten minimalen Wertebereich für zukünftig zu belegende Variablen garantieren. Eine Erweiterung ist die **weighted-max-domain-size**-Heuristik (WMD), die in Fällen von gleich großen minimalen Wertebereichen den Wert auswählt, der weniger Variablen mit dem minimalen Wertebereich erzeugt. Somit werden in Konfliktfällen mehr Variablen mit größeren Wertebereichen ermöglicht, anstatt eine zufällige Auswahl vorzunehmen.

Die letzte hier erwähnte Werteordnungsheuristik ist die **point-domain-size**-Heuristik (PDS), die Punkte für jeden Wert der aktuell zu belegenden Variable vergibt (vgl. Frost und Dechter 1995, S. 575). So können z. B. 8 Punkte für jeden zukünftigen Wertebereich mit nur einem Element, 4 Punkte für zukünftige Wertebereiche mit nur 2 Elementen, 2 Punkte für zukünftige Wertebereiche mit 3 Elementen und 1 Punkt für zukünftige Wertebereiche mit 4 Elementen vergeben werden. Der Wert, der die geringste Punktsumme aufweist, wird demnach von der Heuristik als erstes zur Belegung der aktuellen Variable ausgewählt.

Frost und Dechter (1995) untersuchen experimentell die genannten LVOs und zeigen die besondere Effizienz der *min-conflicts*-Heuristik in Form von FC-CBJ-`dom+deg-mc` auf. Die Anwendung der MC-Heuristik verhält sich demnach ähnlich dem *look-ahead* von FC. Da FC inkonsistente Werte (temporär) aus den Wertebereichen zukünftig zu belegender Variablen entfernt, kann es auch als einfache Form von Werteordnung betrachtet werden (vgl. Frost und Dechter 1995, S. 572). MC hingegen ordnet auch die Werte entsprechend ihrer Konflikte, die Teil einer Lösung sein können und ist so gesehen eine weitergehende Form von *look-ahead*. Analog werden von Bessière und Régin (1996) zur Effizienzsteigerung die Kombinationen MAC-`dom+deg-mc` bzw. MAC-`dom/deg-mc` erfolgreich eingesetzt.

Werteordnungsheuristiken sind stark problemabhängig und weit weniger häufig vertreten als Variablenordnungsheuristiken (vgl. Güsgen 2000, S. 278). Da es u. U. sehr viele Konsistenztests erfordert, um festzustellen welche Werte geeigneter sind als andere zu einer konsistenten Lösung beizutragen, lohnt sich dieser Overhead für gewöhnlich bei einfachen Problemen nicht. Der Einsatz kann allerdings bei sehr umfangreichen CSPs das

[68] Für den speziellen Fall, dass ein BJ-Suchverfahren zum Einsatz kommt, wird von Frost und Dechter (1995, S. 577) gezeigt, dass LVO helfen kann die Unerfüllbarkeit eines CSPs zu ermitteln. Konsistente Teillösungen können hier effizienter ermittelt und ggf. durch BJ übersprungen werden.

Auffinden einer ersten Lösung stark beschleunigen. Je mehr Variablen ein CSP umfasst, umso lohnenswerter ist demnach der Einsatz von LVO, und je kleiner die Wertebereiche der Variablen sind, umso mehr Variablen sind notwendig, um den Overhead von Werteordnungsheuristiken aufzuwiegen (vgl. Frost und Dechter 1995, S. 575 u. 578).

5.2.6 Behandlung von höherwertigen Constraints

Die meisten Algorithmen zur Behandlung von Constraints beschränken sich auf Probleme, die ausschließlich unäre und binäre Constraints beinhalten, sog. *binäre CSPs* (vgl. Definition 5.2.3, S. 85). Binäre Constraints sind ausreichend, solange CSPs als ein „akademisches Problem“ betrachtet werden (n-Damen-Problem, Zebra-Problem, etc.). Da sich jedes *allgemeine CSP* mit polynomialen Aufwand in ein äquivalentes binäres CSP umwandeln lässt, wurde dies häufig zum Anlass genommen, sich aus Gründen der Komplexität auf binäre Constraints zu beschränken (vgl. Barták 2001, S. 8). Neue Ideen und Techniken lassen sich so wesentlich einfacher entwickeln und vorstellen. Zudem haben binäre CSPs den Vorteil, dass sich z. B. Kantenkonsistenz sehr effizient erreichen lässt (vgl. Mackworth und Freuder 1985, S. 66 ff.). In der Folge beschäftigen sich die meisten Arbeiten ausschließlich mit binären CSPs, was dazu führte, dass viele Verfahren nicht für den allgemeinen Fall auf Constraints mit höherer Stelligkeit erweitert wurden (vgl. Bessière 1999, S. 24). In realen Anwendungsdomänen ist es jedoch häufig erforderlich, höherwertige Constraints mit drei oder mehr Variablen, sog. *n-äre* Constraints, innerhalb von allgemeinen CSPs zu betrachten.[69] Die Darstellung der zugehörigen Constraint-Netze kann nicht mehr als binäre Constraint-Graphen erfolgen, sondern in Form von Hypergraphen, in denen Constraints als Hyperkanten über n Knoten repräsentiert werden.

Die Verarbeitung höherwertiger Constraints ist auf unterschiedliche Arten möglich: Zum einen lassen sich, wie bereits angesprochen, n-äre Constraint-Netze in äquivalente binäre Constraint-Netze umwandeln. Zum anderen gibt es Verfahren, die allgemeine CSPs direkt behandeln können. Da die Konvertierung in binäre Constraint-Netze für umfangreiche *real-world*-Probleme aufgrund erhöhter Zeitkomplexität und höherem Speicherbedarf z. T. sehr aufwendig bzw. nicht anwendbar ist,[70] wird in solchen Fällen einer effizienten, direkten Behandlung umso größere Bedeutung beigemessen. Zur Gewährleistung der Effizienz von Suchverfahren spielen allerdings Filtertechniken eine zentrale Rolle, deren Anwendung wiederum in allgemeinen CSPs wesentlich mehr Aufwand verursacht als in binären CSPs, und bei denen für den allgemeinen Fall ein deutliches Hervortreten von algorithmischen Schwachstellen zu beobachten ist. Für viele CSPs kann deshalb die Modellierung des Problems entscheidend sein: Wie ein Problem als Constraint-Netz modelliert wurde, kann wesentlich die Effizienz des Lösungsverfahrens beeinflussen. Selbst einfache Probleme lassen sich auf viele unterschiedliche Arten modellieren. Dies macht es letztendlich erforderlich, die „beste“ Repräsentation des Originalwissens bzgl. des eingesetzten Lösungsverfahrens zu finden, was einen Kompromiss zwischen der Stelligkeit der Constraints und der Effizienz des Lösungsverfahrens einschließt (vgl. Bessière 1999, S. 25; Smith et al. 2000, S. S.187).

[69] Engl. auch *general CSPs*, *non-binary CSPs* oder *high order CSPs* genannt.

[70] Bei einer Umformung müssen i. d. R. neue Variablen und neue Constraints eingeführt werden.

5.2.6.1 Direkte Verarbeitung allgemeiner CSPs

Eine Möglichkeit, CSPs mit höherwertigen Constraints zu verarbeiten, besteht in der Anwendung der in Abschnitt 5.2.3.5 auf Seite 103 vorgestellten Synthese-Algorithmen zur Herstellung von k-Konsistenz, was allerdings aus Komplexitätsgründen nur für die wenigsten Problemstellungen in Frage kommt.

Backtracking-basierte Suchverfahren lassen sich i. A. unkompliziert auch auf n-äre Constraints anwenden, allerdings steigt der Aufwand eine oder gar alle Lösungen zu finden sehr stark an, da selbst für scheinbar einfache Constraints eine sehr große Anzahl Wertekombinationen auf das Bilden einer möglichen Lösung getestet werden müssen (vgl. Marriott und Stuckey 1999, S. 97). Filtertechniken zur Problemreduktion sind daher umso notwendiger, damit der nachfolgend bzw. parallel durchgeführte Suchvorgang vereinfacht wird. Einige Konsistenztechniken für binäre Constraints sind in der Vergangenheit für die direkte Behandlung von n-ären Constraints erweitert worden. Das am häufigsten eingesetzte Konsistenzverfahren ist die Kantenkonsistenz, die bereits sehr früh zur *Hyperkantenkonsistenz* generalisiert wurde. Beim populären Forward Checking dagegen gibt es mehrere Definitionen für die Behandlung von höherwertigen Constraints, die jeweils unterschiedliche Konsistenzgrade und Verarbeitungsverfahren nach sich ziehen (vgl. Bessière et al. 1999b, c, 2002).[71] Ursprünglich wurde FC von Van Hentenryck (1989, S. 60 ff.) für die Bearbeitung von n-ären CSPs verallgemeinert. Analog zur binären Form, wird demnach Forward Checking nur dann vorgenommen, wenn in einem Constraint bereits alle bis auf eine Variable mit einem Wert belegt sind. Die verbleibende Variable ist in diesem Fall *forward checkable*.

Für die Anwendung von Kantenkonsistenz gibt es wie für Backtracking nur eine einzige, eindeutige Definition für die Behandlung von n-ären Constraints (vgl. Mohr und Masini 1988):

Definition 5.2.12 (Hyperkantenkonsistenz/generalized arc consistency, GAC) *Gegeben sei ein CSP auf den Variablen $v_1, \ldots, v_n$ mit den Wertebereichen $D_1, \ldots, D_n$. Sei $C_{j,\ldots,k}$ ein Constraint mit der Relation $R_{j,\ldots,k}$, welche für die Variablen $v_j, \ldots, v_k$ mit $j, \ldots, k \in \{1, \ldots, n\}$ gültige Belegungen spezifiziert. Eine Belegung ist kantenkonsistent, gdw. $\forall v_i \in V$, $\forall d_i \in D_i$, $\forall R_{j,\ldots,k}$, die v_i beschränken, $\forall v_j, \ldots, v_k$, $\exists d_j, \ldots, d_k$, so dass $R_{j,\ldots,k}(d_j, \ldots, d_k)$ erfüllt ist.*

Dies bedeutet lediglich, dass für jede Belegung d_i von jedem Knoten v_i gültige Belegungen in den Domänen der Variablen $v_j, \ldots, v_k$ existieren, so dass jedes n-äre Constraint auf diesen Variablen erfüllt wird. Ein Constraint-Netz ist hyperkantenkonsistent, wenn alle Hyperkanten (Constraints) hyperkantenkonsistent sind. Die Hyperkantenkonsistenz dehnt somit die Konsistenzbedingung der Kantenkonsistenz aus, damit diese auf jedes Constraint unabhängig von der Anzahl der Variablen angewendet werden kann. Allerdings können hierbei Konsistenzprüfungen selbst für einfache Constraints mit mehr als zwei Variablen

[71] Ebenso wie FC, PLA und FLA bzw. MAC unterschiedliche Grade Kantenkonsistenz erreichen, werden von Bessière et al. (1999b, c, 2002) eine Reihe von unterschiedlichen FC-Algorithmen diskutiert, die z. T. die im Folgenden vorgestellten Algorithmen zur Herstellung von Hyperkantenkonsistenz nutzen, und dabei unterschiedliche Grade lokaler Konsistenz erreichen.

wiederum sehr aufwendig werden, da der Aufwand zur Herstellung von Hyperkantenkonsistenz bereits für ein einzelnes Constraint NP-hart ist (vgl. Marriott und Stuckey 1999, S. 97).[72]

Aufgrund dieser Bedingungen existieren nur wenige bekannte Algorithmen zur Herstellung von Hyperkantenkonsistenz. *Alan K. Mackworth* hat in seiner Arbeit zur Interpretation von handgezeichneten Skizzenkarten (engl. *sketch maps*) mittels Constraints den Algorithmus `AC-3` zu einem Algorithmus namens `CN` zum Lösen n-ärer Constraints verallgemeinert (vgl. Mackworth 1977b). Wie `AC-3` besitzt `CN` aufgrund wiederholt vorkommender Konsistenztests für bereits getestete Wertekombinationen eine weniger gute *worst-case* Zeitkomplexität: $O(er^2d^{r+1})$, wobei r die maximale Stelligkeit der Constraints ist (siehe Tabelle 5.1, S. 106). Infolgedessen sollte `CN` nur auf Constraints mit relativ geringer Stelligkeit und kleinen Wertebereichen angewendet werden (vgl. Bessière und Régin 1997, S. 399).

Roger Mohr und *Gérald Masini* erweitern den Algorithmus `AC-4` und stellen `GAC-4` (*Generalized* `AC-4`) vor, mit dem sich Kantenkonsistenz auch für n-äre Constraint-Netze herstellen lässt (vgl. Mohr und Masini 1988). Ebenso wie `AC-4` ist `GAC-4` ein optimaler Algorithmus, und besitzt daher gegenüber `CN` eine verbesserte *worst-case* Zeitkomplexität: $O(ed^r)$. `GAC-4` hat allerdings wie `AC-4` den Nachteil der im Durchschnitt weniger guten Zeitkomplexität und einer sehr hohen Platzkomplexität. Letzteres entsteht, da `GAC-4` ebenso wie `AC-4` auf eine speicherintensive, extensionale Repräsentation der Constraints durch Wertetupel in Listen mit *support*-Einträgen angewiesen ist (vgl. Abschnitt 5.2.3.3, S. 96). `GAC-4` sollte daher ausschließlich auf CSPs angewendet werden, die nur eine geringe Anzahl erlaubter Wertetupel aufweisen und die entsprechend stark beschränkenden Constraints bzw. kleine Wertebereiche beinhalten.

Zuletzt haben *Christian Bessière* und *Jean-Charles Régin* den Ansatz für einen Algorithmus `GAC-Schema` vorgestellt, der den Algorithmus `AC-7` für n-äre Constraints nutzbar macht und deshalb z. T. auch `GAC-7` genannt wird (vgl. Bessière und Régin 1997). `GAC-Schema` kann sowohl auf extensionale als auch auf intensionale, algebraische Constraint-Repräsentationen angewendet werden, und ist bei deutlich reduzierter Speicherintensität ebenso wie `GAC-4` ein optimaler Algorithmus.[73] Eine experimentelle Evaluierung von `GAC-Schema` im Vergleich zu `CN` findet sich in der Arbeit von Sillito (2000).[74] Mit `GAC-Schema` ist es aufgrund der Fähigkeit n-äre Constraint-Relationen zu verarbeiten möglich, mit relativ geringem Einsatz höhere Konsistenz als Kantenkonsistenz zu erzielen, indem Konjunktionen von (binären) Constraints betrachtet werden. Diese Art lokale Konsistenz wird *conjunctive consistency* genannt (vgl. Bessière und Régin 1998). In Abhängigkeit von der Problemstellung kann der Mehraufwand zur Konsistenzherstellung einen Effizienzgewinn bedeuten oder nicht. Eine einfache Heuristik, mit der die Obergrenzen für die Anzahl der jeweils miteinander verknüpften Constraints festgelegt werden können, wird von Katsirelos und Bacchus (2001) vorgestellt.

[72] Die *worst-case* Zeitkomplexität von Algorithmen zur Herstellung von Hyperkantenkonsistenz besitzt ein exponentielles Wachstum bezogen auf die maximale Stelligkeit der Constraints.

[73] `GAC-Schema` kommt z. B. im ILOG Solver zum Einsatz (vgl. Abschnitt 4.5.2, S. 69).

[74] Der Algorithmus `CN` wird von Sillito (2000) mit `GAC-3` benannt.

5.2.6.2 Binärisierung von Constraints

Statt der direkten Verarbeitung n-ärer Constraints ist es möglich, eine Transformation in binäre Constraints vorzunehmen. Dies kann vorteilhaft sein, da für binäre Constraints eine Vielzahl an Algorithmen und Heuristiken bekannt sind, die z. T. für n-äre Constraints bisher nicht anwendbar sind.[75] Durch die einfache Projektion eines n-ären Constraints auf Paare von binären Constraints auf denselben Variablen wird allerdings die ursprüngliche Lösungsmenge lediglich durch eine Obermenge approximiert (vgl. Barták 2001, S. 8). Nur für bestimmte n-äre Constraint-Arten, wie z. B. dem `all-different`-Constraint ($v_1 \neq v_2 \neq \ldots \neq v_k$), erhält man eine Menge von binären Constraints mit derselben Lösungsmenge wie das ursprüngliche CSP. Um ein äquivalentes Constraint-Netz mit derselben Lösungsmenge wie das originale CSP zu erhalten, ist i. A. die Einführung zusätzlicher Variablen notwendig. Diese neuen Variablen umfassen jeweils die Menge der durch ein n-äres Constraint beschränkten Variablen in Form des kartesischen Produkts der jeweiligen Wertebereiche.[76] Ein beliebiges n-äres Constraint lässt sich nun durch eine solche umfassende Variable (engl. *encapsulated variable*) darstellen, indem dessen Wertebereich um die nicht erlaubten Wertekombinationen reduziert wird.

Beispiel 5.2.8 *Das n-äre Constraint*

$$C_{1,2,3} : v_1 + v_2 = v_3$$

mit den Wertebereichen

$$v_1 : \{1, 2\}, \quad v_2 : \{3, 4\}, \quad v_3 : \{5, 6\}$$

lässt sich auf diese Weise durch die umfassende Variable

$$u : \{(1, 4, 5), (2, 3, 5), (2, 4, 6)\}$$

repräsentieren, wobei sich die erste Stelle dieser Tripel jeweils auf einen Wert von v_1, die zweite auf v_2 und die dritte entsprechend auf v_3 bezieht (vgl. Güsgen 2000, S. 271).

Diese Variablen bzw. Lösungen für einzelne n-äre Constraints werden anschließend mittels binärer Constraints verknüpft, um die Gesamtlösungen für das CSP berechnen zu können. Es werden zwei unterschiedliche Arten der Konvertierung von n-ären nach binären Constraint-Netzen unterschieden: die *Hidden-Variable*-Repräsentation und die *Dual-Graph*-Repräsentation, die im Folgenden kurz skizziert werden.

Hidden-Variable-Repräsentation: Für den Ursprung der *Hidden-Variable*-Repräsentation wird von Rossi et al. (1989, S. 3) auf die Arbeit von Peirce (1933) verwiesen. Hier wurde von *Charles S. Peirce* auf dem Gebiet der philosophischen Logik bereits

[75] Beispielsweise MFC und LAC (vgl. Bacchus und van Beek 1998, S. 311).

[76] Bei einem FCSP sind die Domänen der Variablen endlich, insofern ist das kartesische Produkt dieser Wertebereiche ebenfalls endlich.

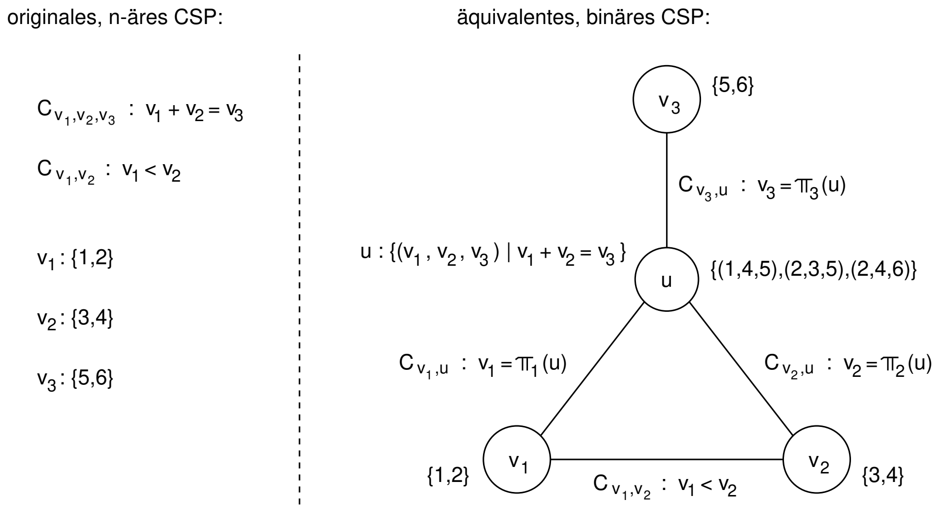


Abbildung 5.19: Hidden-Variable-Repräsentation n-ärer Constraints (vgl. Barták 2003)

formal nachgewiesen, dass binäre Relationen grundsätzlich dieselbe Ausdrucksstärke wie n-äre Relationen besitzen.[77] Auf der Methode von Peirce aufbauend, zeigt Dechter (1990b), wie jede n-äre Relation mittels binärer Relationen und Nutzung von umfassenden Variablen, in diesem Fall *Hidden*-Variablen genannt, mit begrenzten Wertebereichen dargestellt werden kann. Bei der *Hidden-Variable*-Repräsentation werden umfassende Variablen und originale Variablen kombiniert. Lediglich die n-ären Constraints werden in umfassende bzw. *Hidden*-Variablen transformiert, an die wiederum die originalen Variablen durch zusätzliche binäre Kompatibilitäts-Constraints, auch *Hidden*-Constraint genannt, gebunden werden. Hierfür wird eine Projektion π eingeführt, welche die Relation $v = i$-tes Argument von u beschreibt:

$$v = \pi_i(u)$$

Die Werte aus der Domäne der Variable v werden auf die i-te Stelle der Tupel aus dem Wertebereich der umfassenden Variable u projiziert (d. h. i ist die Position von v in u). Ein Beispiel für ein derartig transformiertes, binäres Constraint-Netz ist in Abbildung 5.19 zu sehen. Von Vorteil ist bei diesem Vorgehen der Erhalt der originalen Variablen. Lösungen können direkt abgelesen werden, ohne dass eine Umsetzung anhand der *Hidden*-Variablen vorgenommen werden muss. Zudem müssen weniger Constraints in der speicheraufwendigen, extensionalen Repräsentation in den umfassenden Variablen kodiert werden, als bei der *Dual-Graph*-Repräsentation (vgl. Barták 2003).

Dual-Graph-Repräsentation: Die Repräsentation als *Dual-Graph* stammt ursprünglich aus dem Bereich der relationalen Datenbanken und wurde von *Rina Dechter*

[77] Indem gezeigt wurde, dass ein Ansatz, der Variablen hinzufügt, um binäre Relationen zu erhalten, mit der originalen Repräsentation logisch äquivalent ist.

und *Judea Pearl* in den CSP-Bereich eingeführt (vgl. Dechter und Pearl 1989).[78] Bei der *Dual-Graph*-Repräsentation wird das umgewandelte, binäre CSP ausschließlich durch umfassende Variablen dargestellt, d. h. auch binäre Constraints werden konvertiert.[79] Ein *Dual Graph* ist dadurch gekennzeichnet, dass die Knoten und Kanten im Gegensatz zur normalen Darstellung, auch *Primal Graph* genannt, vertauscht sind (vgl. Dechter und Pearl 1989, S. 355; Stergiou und Walsh 1999, S. 164). Im Fall der Binärisierung eines n-ären CSP sind die Constraints extensional in den Knoten, d. h. in den Wertebereichen der umfassenden Variablen, hier Dual-Variablen genannt, enthalten. Die Kanten hingegen repräsentieren die gemeinsamen Variablen der Constraints (vgl. Rossi et al. 1989, S. 3; 1990, S. 554). Während ein *Primal Graph* für den allgemeinen Fall auch n-äre Constraints enthalten kann, ist sein entsprechender *Dual Graph* immer ein binärer Graph (vgl. Rossi et al. 1989, S. 21; 1990, S. 555). Wird eine Variable ursprünglich von mehreren Constraints beschränkt, so werden die entsprechenden Dual-Variablen in der *Dual-Graph*-Repräsentation über ein binäres Kompatibilitäts- bzw. Dual-Constraint miteinander verbunden. Hierfür wird wiederum die Projektion π genutzt, in diesem Fall zur Beschreibung der Relation i-tes Argument von $u_1 = j$-tes Argument von u_2:

$$\pi_i(u_1) = \pi_j(u_2)$$

Diese Relation verknüpft somit jeweils die i-te mit der j-ten Stelle der Wertetupel zweier umfassender Variablen. Das vormalig bereits vorgestellte n-äre Constraint-Netz ist in binärer *Dual-Graph*-Repräsentation in Abbildung 5.20 auf der gegenüberliegenden Seite dargestellt. Dieses Constraint-Netz ist überschaubarer als das in Abbildung 5.19 auf der vorherigen Seite dargestellte transformierte CSP, allerdings müssen für die Bestimmung einer Lösung des ursprünglichen CSPs die Variablenbelegungen aus den Dual-Variablen extrahiert und den originalen Variablen zugewiesen werden (vgl. Barták 2003). Die hierfür benötigten Informationen werden separat zwischengespeichert. Das resultierende Constraint-Netz ist eine vollständig extensionale Repräsentation der Constraints, da alle für einzelne Constraints gültigen Kombinationen in den umfassenden Variablen enthalten sind.

Von *Kostas Stergiou* und *Toby Walsh* wird außerdem eine Kombination der *Hidden-Variable*- und *Dual-Graph*-Repräsentation eingeführt, die *Double*-Repräsentation (vgl. Stergiou und Walsh 1999). Sie vereint sowohl die Vor- als auch die Nachteile der beiden vorher genannten Repräsentationen, da beide vollständig enthalten sind. Es existieren sowohl die originalen Variablen als auch die Dual-Variablen, und neben den Dual-Constraints zwischen den Dual-Variablen sind ebenfalls die *Hidden*-Constraints zwischen den originalen und den Dual- bzw. *Hidden*-Variablen enthalten.

[78] Ein inkrementeller Algorithmus, der diese Methode zum Auffinden aller Lösungen innerhalb eines CSPs nutzt, wurde von Freuder (1978) bereits mit seinem Synthese-Algorithmus (vgl. Abschnitt 5.2.3.5, S. 105) vorgestellt (vgl. Bacchus und van Beek 1998, S. 311).

[79] Tatsächlich lässt sich die *Dual-Graph*-Repräsentation aufbauend auf der *Hidden-Variable*-Repräsentation erzeugen, indem die zusätzlichen Dual-Variablen und Dual-Constraints erzeugt und die originalen Variablen entfernt werden (vgl. Bacchus et al. 2002, S. 8; Stergiou und Walsh 1999, S. 164 f.).

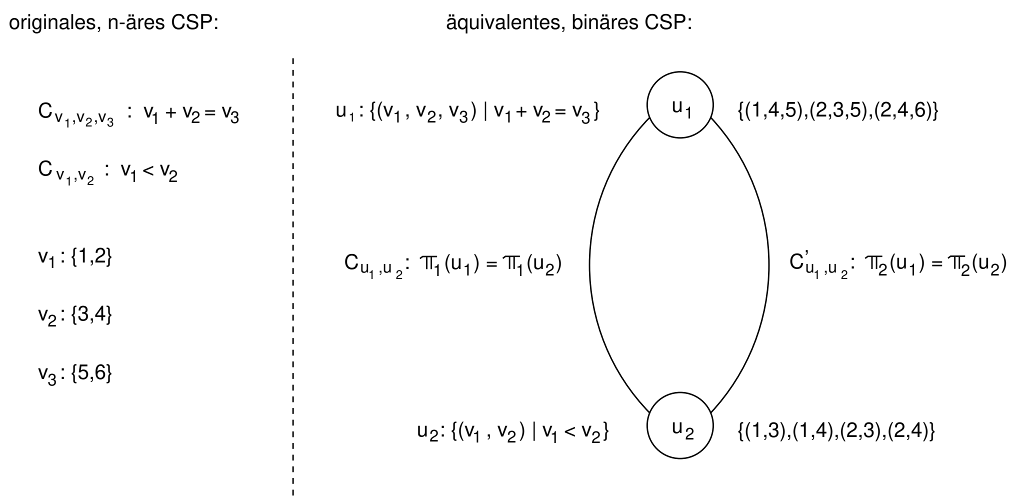


Abbildung 5.20: Dual-Graph-Repräsentation n-ärer Constraints (vgl. Barták 2003)

Ebenfalls von Stergiou und Walsh (1999), aber auch von Bacchus et al. (2002), wird festgestellt, dass Kantenkonsistenz angewandt auf ein Constraint-Netz in *Dual-Graph*-Repräsentation einen höheren Grad lokaler Konsistenz erreicht als Hyperkantenkonsistenz auf der ursprünglichen n-ären Form, da bedingt durch die extensionale Repräsentation mehr inkonsistente Wertekombinationen aus den Domänen der Variablen herausgefiltert werden können.[80] Kantenkonsistenz angewandt auf ein Constraint-Netz in *Hidden-Variable*-Repräsentation hingegen ist äquivalent zur Hyperkantenkonsistenz in der n-ären Form.[81]

Die Frage, ob die Transformation eines n-ären CSP in eines binäres CSP sinnvoll ist, muss in Abhängigkeit von der Problemstellung und von der Modellierung des Problems beantwortet werden. Idealerweise ist ein CSP in der natürlichsten Art und Weise modelliert, und der Rechner löst es möglichst auf die effizienteste Art und Weise (vgl. Bacchus und van Beek 1998, S. 312). In diesem Fall muss der Wissensingenieur die Entscheidung treffen, und in der Wissensbasis zusammen mit der Problemstellung definieren, ob eine Konvertierung vorgenommen werden soll oder nicht. Wobei für die meisten Probleme die n-äre Repräsentation die effizientere bzgl. der Anwendung von Lösungsverfahren ist. Für eine Klasse von stark einschränkenden Constraints, kann die binäre Transformation jedoch wesentlich effizienter sein (vgl. Bacchus und van Beek 1998; Bacchus et al. 2002).

Aufgrund der vollständig extensionalen Darstellung im *Dual-Graph* lohnt sich diese Repräsentation umso mehr, je weniger erfüllbare Wertetupel die Constraints aufweisen. Je größer die Menge der zu verwaltenden Wertetupel ist, umso schlechter verhält sich

[80] Auch wenn Hyperkantenkonsistenz für ein n-stelliges Constraint hergestellt wurde, sind anschließend mit den Wertebereichen der beteiligten Constraint-Variablen i. d. R. Wertekombinationen möglich, die keine Lösung des Constraints darstellen. Dies ist bei einer extensionalen Constraint-Repräsentation nicht möglich, da hier ausschließlich gültige Kombinationen vorausgesetzt werden. So lässt sich in bestimmten Situationen in der *Dual-Graph*-Repräsentation z. B. feststellen, dass ein CSP nicht lösbar ist, während dies mit Hyperkantenkonsistenz in der n-ären Repräsentation nicht gelingt (vgl. Bacchus et al. 2002, S. 13).

[81] Wenn die zusätzliche Reduktion der ausschließlich erlaubten Wertekombinationen in den *Hidden*-Variablen nicht beachtet wird (vgl. Stergiou und Walsh 1999, S. 166).

das Laufzeitverhalten der Lösungsverfahren, da umso mehr Konsistenztests durchgeführt werden müssen, die u. U. in der n-ären Repräsentation redundant sind (vgl. Bacchus und van Beek 1998, S. 314). Ähnlich, jedoch nicht ganz so ausgeprägt, gilt dies ebenfalls für die *Hidden-Variable*-Repräsentation, da hier ausschließlich die n-ären Constraints in *Hidden*-Variablen transformiert werden, und der Zugriff auf die originalen Variablen erhalten bleibt. Bacchus und van Beek (1998) stellen zudem einen eigens für die *Hidden-Variable*-Repräsentation angepassten Algorithmus `FC`$^+$ vor, der eine höhere Reduktion erreicht, als der normale FC-Algorithmus. In der *Double*-Repräsentation können flexibel die Vorteile der einen wie der anderen Repräsentation genutzt werden. Weiterhin lassen sich bei Bedarf unterschiedliche hybride Repräsentationen erzeugen, die u. U. spezielle Constraints nur in der einen oder anderen Repräsentation enthalten, um z. B. die Nachteile der extensionalen Darstellung für bestimmte Constraints zu umgehen (vgl. Smith et al. 2000; Stergiou und Walsh 1999). Dies erfordert jedoch zusätzlichen Anpassungsaufwand des Lösungsverfahrens an die Problemstellung.

5.2.6.3 Bounds Consistency

Von Marriott und Stuckey (1999, S. 97 ff.) wird vorgeschlagen, n-äre Constraints durch Einführung einer neuen Konsistenzart zu behandeln: *bounds consistency*.[82] Um *bounds consistency* herzustellen, werden lediglich die Ober- und Untergrenzen der Wertebereiche propagiert. Für viele CSPs mit finiten Domänen würde dies eine Approximation der Wertebereiche der Constraint-Variablen an Intervalle mit der Ober- und Untergrenze selbiger Wertebereiche bedeuten.[83] Bei einer solchen Annäherung werden Ungenauigkeiten in Kauf genommen, was dazu führt, dass die Vollständigkeit und Korrektheit der Lösung bezogen auf das ursprüngliche CSP nicht gewährleistet werden kann.[84] Um große Wertebereiche zu verarbeiten, kann dies allerdings die einzig effiziente Möglichkeit sein, und z. B. als Preprozessing in Kombination mit anderen Konsistenz- und Suchverfahren zum Einsatz kommen.

Mittels *bounds consistency* wie von Marriott und Stuckey (1999) beschrieben, lassen sich zwar n-äre Constraints behandeln, um dies zu erreichen müssen allerdings für jede primitive Constraint-Art, d. h. für jede Form einer möglichen Gleichung bzw. Ungleichung, die Propagationsregeln in Form von *global constraints* explizit definiert und implementiert werden (vgl. Abschnitt 4.1, S. 53). Eine Umsetzung dieser Art *bounds consistency* würde eine nur sehr eingeschränkte Verwendung ermöglichen. Wenn diese Art Konsistenz zur Einschränkung von Wertebereichen angewendet wird, sollten allgemeinere Konzepte zum Einsatz kommen, welche die Behandlung beliebiger algebraischer Constraints ermöglichen. Weiterführendes zur Propagation von Intervallgrenzen ist dem folgenden Abschnitt 5.3 zu entnehmen.

[82] *bound* (engl.): Grenze, Schranke

[83] Die von Marriott und Stuckey (1999) definierte *bounds consistency* ist auf arithmetische Constraints mit finiten Integer-Domänen beschränkt.

[84] Das heißt die Lösungsmenge kann ggf. Werte enthalten, die nicht Teil einer potentielle Lösung sind und zudem nicht in dem ursprünglichen CSP enthalten waren. Entsprechend könnten in der Konsequenz globale Lösungen mit Werten generiert werden, die sich nicht im Lösungsraum des ursprünglichen CSP befinden und demnach keine Lösung für selbiges darstellen können.

5.3 Intervall Constraint Satisfaction Probleme

Neben klassischen CSP über finite Domänen stellt sich für die Constraint-Verarbeitung in EngCon das Problem der Behandlung von reellwertigen algebraischen Constraints. Die Wertebereiche der Constraint-Variablen werden hier als Ober- und Untergrenzen von reellwertigen Intervallen definiert, zwischen denen sich unendlich viele, nicht abzählbare Elemente befinden (vgl. Benhamou 2001; van Emden 2002). Diese Wertebereiche werden deshalb auch *kontinuierlich* genannt. Der Einsatz von Intervall-Constraints bietet sich für eine Reihe von Szenarien an:

- Zur Behandlung unscharfer Informationen, d. h. wenn das Wissen über Parameter nur in Form eines Werteintervalls bekannt ist.
- Wenn die Aufzählung aller Lösungen nicht möglich ist, da unendlich viele existieren.
- Unterbestimmte Systeme, für die mehr als eine Lösung bzw. ein Lösungsintervall existiert, können besser behandelt werden. Beispiel: optimaler Drehzahlbereich für einen Motor ist $7000 \leq x \leq 9000$, d. h. $x \in [7000, 9000]$.
- Statt kombinatorischer Verfahren über einzelne Werte, wird die gleichzeitige Einschränkung von unendlich vielen Werten ermöglicht.

Im Gegensatz zu diskreten, endlichen Wertebereichen, für die Konsistenz- und Lösungsalgorithmen die einzelnen Werte in den Domänen der Constraint-Variablen mittels kombinatorischer Methoden aufzählen und auf Zugehörigkeit zu den Constraint-Relationen überprüfen können, lässt sich im Fall von reellwertigen Intervallen nicht für jeden einzelnen Wert bestimmen, ob er als konsistente Belegung geeignet ist. Während CSPs über endliche Domänen zur Klasse der NP-vollständigen Probleme zählen, sind CSPs über unendliche Domänen unentscheidbar (vgl. Güsgen 2000, S. 269). Daher werden hier durch Konsistenz- und Suchverfahren lediglich die Ober- und Untergrenzen der Intervalle überprüft bzw. angepasst, so dass inkonsistente Werte ausgeschlossen werden. Dieses Framework wird *Interval Constraint Satisfaction Problem* (ICSP) (vgl. Chopra 1997; Hyvönen 1989, 1991, 1992; Yang und Yang 1997) genannt, ist aber auch unter den Namen *Numerical Constraint Satisfaction Problem* (NCSP) (vgl. Benhamou 2001; Lebbah und Lhomme 1998, 2002; Lhomme 1993; Ruttkay 1998; Silaghi et al. 2001; Vu et al. 2002, 2003c) und *Continuous Constraint Satisfaction Problem* (CCSP) (vgl. Benhamou 2001; Cruz und Barahona 2003; Haroud und Faltings 1994; Sam-Haroud und Faltings 1996a, b; Sam-Haroud 1995) bekannt. Für gewöhnlich werden reellwertige Intervalle vorausgesetzt, das Konzept lässt sich jedoch auch auf andere Domänen, wie diskrete Integer-Intervalle, verallgemeinern (vgl. Benhamou 2001; Chopra 1997; Lhomme 1993; Yang und Yang 1997).

Definition 5.3.1 (Intervall Constraint Satisfaction Problem, ICSP)
Ein Intervall Constraint Satisfaction Problem wird durch ein Tripel (V, I, C) *beschrieben. Neben den die Menge der Constraints* C *beschränkenden Variablen* $V = \{v_1, \ldots, v_n\}$ *wird eine Menge von Intervallen* $I = \{I_1, \ldots, I_n\}$ *als Wertebereiche der Variablen mit* $\{v_1 : I_1, \ldots, v_n : I_n\}$ *definiert.* C *ist die endliche Menge von Constraints* $C_j(V_j)$, $j \in$

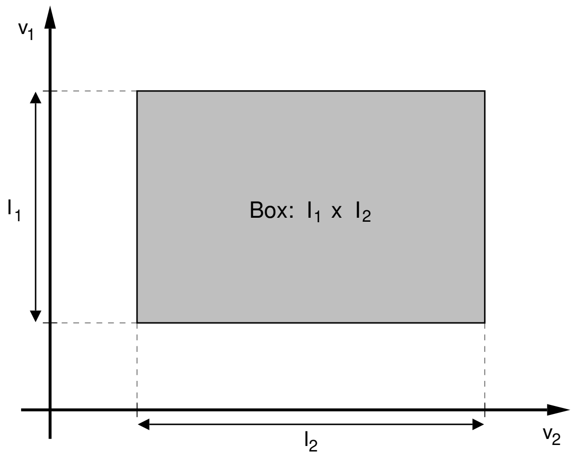


Abbildung 5.21: Lösungsraum für ein ICSP mit den Variablen v_1 und v_2

$\{1,\ldots,m\}$. *Jedes Constraint* $C_j(V_j)$ *setzt eine Teilmenge* $V_j = \{v_{j_1},\ldots,v_{j_k}\} \subseteq V$ *zueinander in Relation, und beschränkt den Lösungsraum der involvierten Variablen auf eine Teilmenge des kartesischen Produkts* $I_{j_1} \times \cdots \times I_{j_k}$ *von k Subintervallen.*

Das kartesische Produkt mehrerer Intervalle wird aus geometrischen Gründen auch kurz *Box* genannt (siehe Abbildung 5.21). Ziel ist es, eine Menge n-stelliger, möglichst *kanonischer* Boxen zu isolieren, die den Lösungsraum des CSP approximieren, ohne gültige Lösungen zu verlieren. Jede n-stellige Box approximiert jeweils eine mögliche Lösung des CSP. Eine Box wird „kanonisch“ genannt, wenn sie aus Intervallen besteht, deren Grenzen entweder jeweils dieselben oder direkt aufeinander folgende Zahlen sind, d. h. wenn sie möglichst punktgenaue Lösungen darstellen. Um dies zu erreichen wird mittels Such- und Konsistenztechniken sowie mathematischer Verfahren durch den Raum navigiert, der durch das kartesische Produkt $I_1 \times \cdots \times I_n$ aufgespannt wird (vgl. Benhamou 2001, S. 45). Für diese Art der Constraint-Propagation, auch Intervallpropagation genannt, gibt es eine Reihe von in den folgenden Abschnitten aufgezeigten Verfahren, die auch unter den Begriffen *bound consistency* und *bound propagation* zusammengefasst werden (vgl. Russel und Norvig 2002, S. 148).

Constraint satisfaction über reellwertige Intervalle wurde erstmals von *John G. Cleary* und *Ernest Davis* betrachtet. Ziel von Cleary (1987) war es, im Rahmen von *Constraint Logic Programming* (CLP) Abhilfe für die Inkorrektheit von Fließkomma-Berechnungen in der Sprache Prolog zu schaffen. Die Bemühungen mündeten in einem neuem Prolog-Dialekt mit dem Namen BNR Prolog und führten seitdem zu einer Reihe weiterer CLP-Sprachen, die über integrierte Constraint-Solver in der Lage sind, reellwertige Intervall-Constraints zu verarbeiten (vgl. Benhamou 2001, S. 46; Sam-Haroud 1995, S. 30).[85] Das Einschränken der Intervallgrenzen wurde von Cleary (1987) mittels einfachem *Domänen-Splitting* und einer Backtracking-Suche auf dem entstehenden Baum durchgeführt (vgl.

[85] Beispielsweise CLP(BNR), Prolog IV, Newton, Numerica, etc. (vgl. Abschnitt 4.5.1, S. 66).

Abschnitt 5.3.2, S. 150). Davis (1987) hingegen adaptierte eine Version des Waltz-Filteralgorithmus für die Behandlung von Intervall-Constraints (vgl. Abschnitt 5.3.3, S. 151). Bei diesen ersten Ansätzen waren für jede Constraint-Art jeweils spezielle Algorithmen notwendig, um eine Einschränkung der Wertebereiche vornehmen zu können.[86] Weiterentwicklungen dieser Verfahren wurden verallgemeinert, so dass beliebige algebraische Constraints verarbeitet werden können (vgl. Benhamou et al. 1994a; Hyvönen 1992; Lhomme 1993).

Den Verfahren zum Auflösen eines ICSP ist gemein, dass mit Hilfe von Intervallmathematik, bzw. sog. Intervallarithmetik, lediglich die Intervallgrenzen der Constraint-Variablen propagiert werden. Intervallarithmetik beschreibt dazu arithmetische Operationen auf Intervallen.

5.3.1 Intervallmathematik

Die zur Auflösung von reellwertigen ICSPs zum Einsatz kommende Intervallmathematik ist ein noch relativ junger Bereich der Numerischen Mathematik, der in den 60er Jahren von Moore (1969) eingeführt wurde. Die Intervallmathematik bietet mit Hilfe der Intervallarithmetik Methoden zur Berechnung der unterschiedlichsten mathematischen Probleme und Anwendungen. So können Relationen und Operationen auf den reellen Zahlen auf Intervallen dargestellt und durchgeführt werden, wodurch Fehler bzw. Ungenauigkeiten, die durch die normalerweise zum Einsatz kommenden Fließkomma-Operationen im Rechner entstehen würden, vermieden werden. Als Lösung wird stets ein paar gesicherter Schranken, obere und untere Intervallgrenzen, berechnet. Man erhält somit nicht nur Näherungslösungen, wie bei herkömmlichen numerischen Verfahren, sondern vielmehr berechnete Fehlerschranken, welche auftretende Rundungsfehler berücksichtigen (vgl. Bauch et al. 1987, S. 7):

Beispiel 5.3.1 *Wird für das Ergebnis z einer Rechnung die gerundete reelle Zahl $x = 0.2834$ verwendet, so ist aus dieser nicht der exakte Wert von z erkennbar. Anhand der Rundungsregeln ist allerdings bekannt, dass z zwischen 0.28335 und 0.28345 liegen muss, d. h. $z \in [0.28335, 0.28345]$.*

Zudem sind praktische Aufgabenstellungen in technischen und naturwissenschaftlichen Anwendungen häufig auf Messergebnisse angewiesen:

Beispiel 5.3.2 *Da kein Messgerät völlig exakte Ergebnisse liefert, müssen Messfehler berücksichtigt werden. Lautet das Ergebnis einer Temperaturmessung $T = 58.0 \pm 0.5°\mathrm{C}$, so gilt für den exakten Wert $T^* \in [57.5, 58.5]$.*

Da in der Praxis häufig gerundete und gemessene Werte Verwendung finden, deren Genauigkeitsintervalle bekannt sind, ging man dazu über, Berechnungen direkt mit diesen Intervallen durchzuführen, woraus die Intervallarithmetik entstand, die sich zum umfassenden Gebiet der Intervallmathematik entwickelt hat.

[86] Vgl. *global constraint*, Abschnitt 4.1, S. 53.

Neben der traditionellen Anwendung der Intervallarithmetik, der Kompensation von Rundungsfehlern, die aufgrund der begrenzten Präzision maschineller Rechenanlagen entstehen, besteht eine weitere Anwendung in der Berechnung von Problemen, deren Daten von Natur aus in Werteintervallen angegeben werden. Dies betrifft z. B. den optimalen Drehzahlbereich eines Motors oder die statischen Belastungen von Bauwerken (vgl. Bauch et al. 1987, S. 29). In der Künstlichen Intelligenz finden sich neben der Konfigurierung viele Problemstellungen, in denen lediglich ungenaues, unscharfes Wissen vorliegt. Für Probleme dieser Art liefert die Intervallmathematik eine natürliche Sprache (vgl. Alefeld und Herzberger 1974; Moore 1969).

5.3.1.1 Intervalle

Ein Intervall ist definiert als ein geordnetes Paar reeller Zahlen $[a_1, a_2]$ mit $a_1 \leq a_2$. Wobei das Intervall $[a, a]$ äquivalent mit der reellen Zahl a ist und deshalb auch *Punktintervall* (vgl. Alefeld und Herzberger 1974; Bauch et al. 1987) oder *entartetes Intervall* (vgl. Moore 1969) genannt wird. Die Werte dieser Tupel stehen jeweils für die untere und obere Schranke eines Intervalls und repräsentieren somit die Teilmenge der reellen Zahlen, die sich innerhalb dieser Grenzen befinden (vgl. Moore 1969, S. 15):

Definition 5.3.2 (Intervalle)
Das Intervall $I_1 = [a_1, a_2]$ besteht aus der Menge der reellen Zahlen $x \in \mathbb{R}$ mit $a_1 \leq x \leq a_2$.

Intervalle sind demnach zusammenhängende Teilmengen in $\mathbb{R}$. Es wird zwischen *abgeschlossenen* bzw. *kompakten*, *offenen* und *halboffenen* Intervallen unterschieden (vgl. Bronstein et al. 1996; Embacher und Oberhuemer 2003). Bei einem abgeschlossenem Intervall $[a_1, a_2]$ gehören die Randpunkte a_1 und a_2 zum Intervall dazu:

$$[a_1, a_2] := \{x \in \mathbb{R} \mid a_1 \leq x \leq a_2\}$$

Während bei einem offenen Intervall $]a_1, a_2[$ die Randpunkte nicht zum Intervall dazu zählen:

$$]a_1, a_2[:= \{x \in \mathbb{R} \mid a_1 < x < a_2\}$$

Halboffene Intervalle $]a_1, a_2]$ und $[a_1, a_2[$ sind jeweils zu einer Seite offen und zur anderen Seite abgeschlossen:

$$\begin{aligned}]a_1, a_2] &:= \{x \in \mathbb{R} \mid a_1 < x \leq a_2\} \\ [a_1, a_2[&:= \{x \in \mathbb{R} \mid a_1 \leq x < a_2\} \end{aligned}$$

Weiterhin können Intervalle nach oben oder nach unten *unbeschränkt* sein, wie z. B. die folgenden Intervalle:

$$\begin{aligned}]-\infty, a] &:= \{x \in \mathbb{R} \mid -\infty < x \leq a\} \\ [a, +\infty[&:= \{x \in \mathbb{R} \mid a \leq x < +\infty\} \end{aligned}$$

Die Menge der reellen Zahlen $\mathbb{R}$ wird demnach auch mit $]-\infty, +\infty[$ bezeichnet.[87]

[87] Anstatt $]a_1, a_2[$, $[a_1, a_2[$, $]a_1, a_2]$ wird in vorwiegend älterer Literatur auch (a_1, a_2), $[a_1, a_2)$, $(a_1, a_2]$ geschrieben. Die modernere Schreibweise mit umgedrehten [ ] zur Kennzeichnung offener Intervallenden

5.3.1.2 Klassische Intervallarithmetik

Die klassische Intervallarithmetik befasst sich ausschließlich mit beschränkten, abgeschlossenen, reellen Intervallen. Auch wenn es wünschenswert wäre, daneben sowohl offene als auch halboffene Intervalle zu verarbeiten, ist dies für Standardoperationen i. d. R. nicht notwendig.[88] Dagegen erlauben abgeschlossene Intervalle bspw. die problemlose Verarbeitung von Punktintervallen als Sonderfall der Intervallarithmetik (vgl. Davis 1987, S. 297).

Die Menge der beschränkten, abgeschlossenen, reellen Intervalle wird mit $\mathsf{I}(\mathbb{R})$ bezeichnet (vgl. Alefeld und Herzberger 1974). Generell existiert für jede Operation der reellen Arithmetik auf $\mathbb{R}$ eine entsprechende Verallgemeinerung bzw. Erweiterung für intervallarithmetische Operationen in $\mathsf{I}(\mathbb{R})$ (vgl. Bauch et al. 1987, S. 8):

Definition 5.3.3 (zweistellige Intervalloperationen)
Sei $\circ \in \{+, -, \times, \div\}$ *eine zweistellige Operation für reelle Zahlen. Dann gelte für die Intervalle* $I_1 = [a_1, a_2]$, $I_2 = [b_1, b_2] \in \mathsf{I}(\mathbb{R})$*:*

$$I_1 \circ I_2 := \{x \circ y \mid a_1 \leq x \leq a_2, b_1 \leq y \leq b_2\}$$

Die Intervalldivision sei nur definiert, falls $0 \notin I_2$*.*

Die allgemeine Definition 5.3.3 für intervallarithmetische Operationen geht auf die mengentheoretische Überlegung zurück, dass Summe, Differenz, Produkt oder Quotient von zwei Intervallen die Menge der Summen, Differenzen, Produkte oder Quotienten von allen Paaren der jeweils in den beiden Intervallen enthaltenen reellen Zahlen sind (vgl. Moore 1969, S. 18). Demnach ist das Ergebnis intervallarithmetischer Grundoperationen wiederum ein Intervall, deren Eckpunkte folgendermaßen formuliert werden (vgl. Alefeld und Herzberger 1974, S. 2):

Definition 5.3.4 (zweistellige intervallarithmetische Grundoperationen)
Es gilt

$$\begin{aligned}
[a_1, a_2] + [b_1, b_2] &= [a_1 + b_1, a_2 + b_2], \\
[a_1, a_2] - [b_1, b_2] &= [a_1 - b_2, a_2 - b_1], \\
[a_1, a_2] \times [b_1, b_2] &= [min(a_1b_1, a_1b_2, a_2b_1, a_2b_2), max(a_1b_1, a_1b_2, a_2b_1, a_2b_2)], \\
[a_1, a_2] \div [b_1, b_2] &= [a_1, a_2] \times [\frac{1}{b_2}, \frac{1}{b_1}], \quad 0 \notin [b_1, b_2], \\
&= [min(\frac{a_1}{b_1}, \frac{a_1}{b_2}, \frac{a_2}{b_1}, \frac{a_2}{b_2}), max(\frac{a_1}{b_1}, \frac{a_1}{b_2}, \frac{a_2}{b_1}, \frac{a_2}{b_2})], \quad 0 \notin [b_1, b_2].
\end{aligned}$$

Das Multiplikationszeichen „×" wurde zur besseren Übersichtlichkeit z. T. weggelassen. Neben diesen zweistelligen Grundoperationen lassen sich auch eine Reihe üblicher einstelliger Intervalloperationen definieren:

hat sich durchgesetzt, um Verwechslungen mit dem geordneten Paar (a_1, a_2) der beiden Zahlen a_1 und a_2 auszuschließen (vgl. Bronstein et al. 1996, S. 226).

[88] Hyvönen (1992, S. 76) empfiehlt ein pragmatisches Vorgehen: Offene Intervallgrenzen lassen sich approximieren, indem durch die Darstellung x^+ und x^- geringfügig größere bzw. kleinere Zahlen als x repräsentiert werden. Das halboffene Intervall $]0, 2]$ entspricht demnach $[0^+, 2]$. Die Darstellung von x^+ und x^- steht dabei für die nächsten, innerhalb der Präzision der aktuellen Implementierung darstellbaren Zahlen.

Definition 5.3.5 (einstellige Intervalloperationen)
Sei $\varphi \in \{\sqrt{\ }, exp, ln, log, abs, sin, cos, \ldots\}$ *eine stetige, einstellige Operation in* $\mathbb{R}$*, dann ist für das Intervall* $I = [a_1, a_2] \in \mathsf{I}(\mathbb{R})$ *durch*

$$\varphi(I) := \{\varphi(x) \mid a_1 \leq x \leq a_2\}$$

eine (zugehörige) einstellige Operation in $\mathsf{I}(\mathbb{R})$ *erklärt.*

Unter Berücksichtigung der Monotonieeigenschaften von φ kann von dieser Definition die folgende Rechenregel wiederum deduktiv abgeleitet werden (vgl. Alefeld und Herzberger 1974, S. 3):

Definition 5.3.6 (stetige, einstellige intervallarithmetische Operationen)
Es gilt

$$\varphi([a_1, a_2]) = [min(\varphi(x)), max(\varphi(x))], \quad a_1 \leq x \leq a_2.$$

Die *natürliche Intervallerweiterung* einer Funktion ist demnach definiert durch die Ersetzung der mathematischen Operatoren durch ihre intervallarithmetischen Äquivalenten. Aus den Definitionen für Intervalloperationen folgt, dass die Intervallarithmetik mit Punktintervallen $[a, a]$ auf die gewöhnliche reelle Arithmetik zurückgeführt werden kann. Die Intervallarithmetik ist daher als eine Erweiterung der reellen Arithmetik zu sehen (vgl. Moore 1969, S. 19). Das Ziel bei der Behandlung von ICSPs ist es, die Intervallgrenzen so weit wie möglich einzuschränken, ohne mögliche Lösungen zu verlieren. Dazu wird für die Bearbeitung auf Rechnersystemen mit Fließkomma-Intervallen das Ergebnis z. B. der Operation $[a_1, a_2] + [b_1, b_2]$ bestimmt, indem die untere Schranke des Intervalls, das Ergebnis der Summe $a_1 + b_1$, abgerundet, und die obere Schranke $a_2 + b_2$ aufgerundet wird (vgl. Moore 1969, S. 16). Indem bei Rechenoperationen die Intervallgrenzen auf diese Weise stets nach außen gerundet werden ist sichergestellt, dass sich die exakte Lösung innerhalb des resultierenden Intervalls befindet (vgl. Bauch et al. 1987, S. 12; Hyvönen 1992, S. 84).

5.3.1.3 Erweiterte Intervallarithmetik

In der klassischen Intervallarithmetik ist die Division durch ein Intervall verboten, welches 0 innerhalb seiner Intervallgrenzen enthält. Um diese Fälle trotzdem korrekt behandeln zu können, muss eine *erweiterte Intervallarithmetik* Anwendung finden. In der Literatur gibt es verschiedene Ansätze für solche Arithmetiken. Eine Erweiterung der klassischen Intervallarithmetik ist die *Kahan-Intervallarithmetik*, in der Rechenvorschriften für die Behandlung der Division durch 0 definiert werden (vgl. Kahan 1968; zit. nach Bauch et al. 1987; Laveuve 1975).

Innerhalb einer erweiterten Intervallarithmetik wird $\mathsf{I}^*(\mathbb{R})$ zum Umgang mit unbeschränkten Intervallen definiert. Da Intervalle als Paare von Intervallgrenzen angegeben werden, kann formuliert werden, dass $\mathsf{I}^*(\mathbb{R})$ aus dem kartesischen Produkt der Menge der unteren Grenzen $\mathsf{L}^*(\mathbb{R})$ und der Menge der oberen Grenzen $\mathsf{U}^*(\mathbb{R})$ gebildet wird:

$$\mathsf{I}^*(\mathbb{R}) = \mathsf{L}^*(\mathbb{R}) \times \mathsf{U}^*(\mathbb{R})$$

Für $x \in \mathsf{L}^*(\mathbb{R})$ gilt dabei, dass $x \in \mathbb{R}$ inklusive $-\infty$, aber ohne $+\infty$, und für $x \in \mathsf{U}^*(\mathbb{R})$ gilt, dass $x \in \mathbb{R}$ inklusive $+\infty$, aber ohne $-\infty$ (vgl. Cleary 1987, S. 128).

Die klassische Intervallarithmetik ist in der Kahan-Intervallarithmetik dementsprechend um die Behandlung von $\pm\infty$ erweitert. Dazu werden neben der Definition einer Division durch 0 ($\frac{1}{0} = \pm\infty$) alle möglichen Fälle bei der Berechnung mit $-\infty$ und $+\infty$ explizit definiert, und hierfür benötigte, zusätzliche (Hilfs-)Intervallarten und die entsprechenden Rechenregeln, die für deren Behandlung erforderlich sind, eingeführt. Das Resultat ist eine Arithmetik, die Regeln zur Berechnung von unbeschränkten, geschlossenen reellen Intervallen aufweist (vgl. Bauch et al. 1987, S. 15 ff.; Laveuve 1975, S. 236 ff.).

5.3.1.4 Konvexität und Diskontinuität

Die bisher betrachteten Intervalle sind ununterbrochen und werden auch *konvexe* Intervalle genannt. Neben konvexen Intervallen können allerdings auch unterbrochene Intervalle auftreten. Diese weisen ähnlich einer unstetigen Funktion Lücken in ihrem Wertebereich auf:

Definition 5.3.7 (konvexes Intervall)
Ein konvexes Intervall ist ein ununterbrochener Wertebereich, definiert durch eine untere und eine obere Schranke. Entsprechend ist ein diskontinuierliches Intervall ein unterbrochener Wertebereich, der aus einzelnen, disjunkten Subintervallen besteht.

Wenn ein Problem ausschließlich Variablen mit konvexen Wertebereichen involviert, kann sich dies positiv auf das Intervall-Lösungsverfahren bzgl. des erreichten Konsistenzgrades und der Vollständigkeit des Ergebnisses auswirken. Analog zu konvexen Intervallen spricht man von konvexen Constraints:

Definition 5.3.8 (konvexes Constraint)
Ein konvexes Constraint besitzt ausschließlich konvexe Intervalle als Ergebnismengen.

Ein konvexes Constraint führt demnach, bedingt durch die Art der Relation, zu konvexen Ergebnisintervallen. Konvexe Constraints werden auch engl. *non-disjunctive* Constraints genannt, da sie die Wertebereiche der Ergebnisintervalle nicht unterbrechen.

Eine wichtige Frage betrifft die Behandlung von diskontinuierlichen Intervallen. Sie treten auf, wenn die Menge der mit einem Constraint konsistenten Werte unterbrochen ist, weil dem Constraint für die Berechnung eines Wertes (1) eine unstetige oder (2) eine mehrwertige Funktion der anderen Werte zugrunde liegt (vgl. Davis 1987, S. 297):

Beispiel 5.3.3 *Für das Constraint $v_1 \times v_2 = v_3$ mit den Wertebereichen $v_2 : [-1, 1]$ und $v_3 : [4, 5]$ ist die Ergebnismenge für v_1 das unterbrochene Intervall $[-\infty, -4] \cup [4, +\infty]$. Dies liegt darin begründet, dass $v_1 = \frac{v_3}{v_2}$ eine unstetige Funktion für $v_2 = 0$ ist.*

Eine mehrwertige Funktion mit einem unterbrochenen Intervall als Ergebnis liegt für das Constraint im folgenden Fall vor:

Beispiel 5.3.4 *Für das Constraint $v_1^2 = v_2$ mit dem Wertebereich $v_2 : [4, 9]$ ergibt sich für $v_1 : [-3, -2] \cup [2, 3]$, weil $v_1 = \sqrt{v_2}$ eine mehrwertige Funktion ist.*

Zur Verarbeitung dieser Ergebnismengen existieren zwei Ansätze: Zum einen kann die Intervallrepräsentation um diskontinuierliche Intervalle, bestehend aus Vereinigungsmengen von Teilintervallen, erweitert werden.[89] Zum anderen ist es möglich, lediglich die konvexe Hülle der Ergebnisintervalle zu betrachten, d. h. das Minimum und das Maximum aller Teilintervalle. Das Ergebnis jeder Operation ist hier ein ununterbrochenes bzw. konvexes Intervall. Dies ist jeweils das kleinste Intervall, welches sämtliche Lösungen umschließt. Der entstehende Informationsverlust wird i. d. R. aufgewogen durch die einfache Verarbeitung konvexer Intervalle (vgl. Davis 1987, S. 297). Viele verfügbare Systeme bzw. Algorithmen implementieren daher das letztere Vorgehen.

5.3.2 Intervall-Splitting

Die vorgestellten intervallmathematischen bzw. -arithmetischen Grundlagen sind Basis der Constraint-Lösungsalgorithmen, die Intervall-Constraints verarbeiten und insbesondere dazu dienen, Intervallgrenzen einzuschränken bzw. Lösungen zu bestimmen. *John G. Cleary* stellte das Konzept des *domain splitting* bzw. *interval splitting* vor, einem Suchverfahren zur Bestimmung von Lösungen in ICSPs (vgl. Cleary 1987). Die Grundidee ist, dass Intervallarithmetik auf Subintervalle angewendet wird, um so engere Grenzen für die Ergebnisintervalle zu erhalten. Ziel von Cleary war es, ein Intervall-Constraint-Framework für CLP-Systeme zur Verfügung zu stellen, dass neben der Intervallrepräsentation ein einfaches Lösungsverfahren anbietet.

Die Suche im Lösungsraum gestaltet sich in der Form, dass Cleary abwechselnd die Wertebereiche der Intervalle zerteilt (engl. *splitting*) und anschließend Berechnungen zum weiteren Einschränken der Intervallgrenzen auf diesen Teilintervallen in separaten ICSPs anstellt (engl. *squeezing*). Die resultierende Backtracking-Suche im Lösungsbaum ist an einem einfachen Beispiel in Abbildung 5.22 auf der gegenüberliegenden Seite dargestellt. Die notwendigen Berechnungen ergeben sich aus ableitbaren Funktionen, was sich bei Grundoperatoren trivial gestaltet.[90] Die sich ergebenen Einschränkungen der Intervallgrenzen werden so lange durchgeführt, bis durch das Suchverfahren eine Lösung gefunden wird. Cleary (1987) beschränkt sich auf Grundoperatoren und ununterbrochene Intervalle.[91]

Weiterentwicklungen und Optimierungen zum Domänen-Splitting betreffen Verbesserungen des Suchvorgangs und der Splitting-Strategie. Jussien et al. (1997) bzw. Jussien und Lhomme (1998) wenden eine dynamische Backtracking-Variante an, um den Splitting-Prozess effizienter zu gestalten (engl. *dynamic domain splitting*). *Dynamic Backtracking* ist eine Variante von abhängigkeitsgesteuertem Backtracking, welche sämtliche getestete Wertekombinationen speichert, anstatt sie, wie klassisches Backtracking, zu verwerfen, wenn eine Inkonsistenz auftritt. Dadurch ist es möglich, allerdings bei zusätzlichem Verwaltungsaufwand und erhöhter Speicherintensität, dynamisch unterschiedliche Stellen im

[89] Die in diesem Fall bei Rechenoperationen auf diskontinuierlichen Intervallen entstehenden überlappenden Teilintervalle werden jeweils zu ununterbrochenen (Teil-)Intervallen zusammengefasst.

[90] Beispielsweise $(v_1 + v_2 = v_3)$, $(v_3 - v_1 = v_2)$, $(v_3 - v_2 = v_1)$.

[91] Cleary nennt die Eigenschaft eines ununterbrochenen Intervalls *interval convexity* und hebt die vereinfachte Berechnung mit diesen hervor (vgl. Cleary 1987, S. 132).

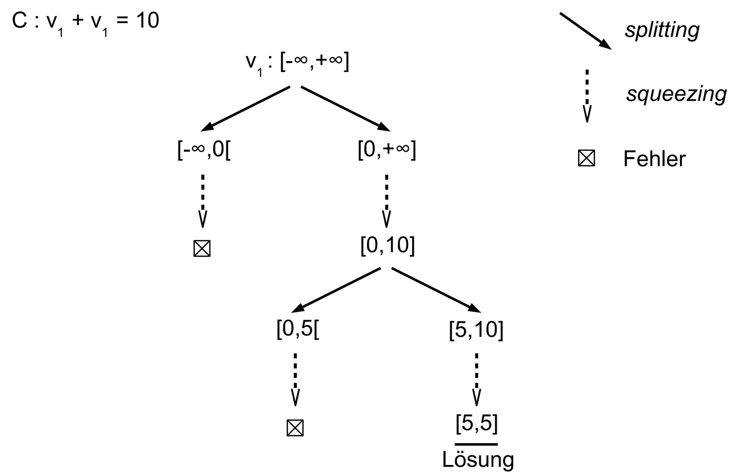


Abbildung 5.22: Einfaches Beispiel für Domänen-Splitting (vgl. Cleary 1987, S. 135)

Suchbaum zu bearbeiten, anstatt diesen, wie beim klassischen Backtracking, ausschließlich von oben nach unten zu durchlaufen (vgl. Ginsberg 1993).

Silaghi et al. (1999) entwickeln ein Verfahren, dass informiert vorgeht[92] und die Splitting-Punkte intelligent auswählt (engl. *intelligent domain splitting*). Benachbarte Regionen, die Lösungen beinhalten, werden auf diese Weise nicht zerteilt, wodurch der Splitting-Prozess effektiver wird, da insgesamt weniger „Splittings" erforderlich sind. Das Verfahren ist allerdings aufgrund der Repräsentation der Wertebereiche auf finite Domänen begrenzt. Um diesen Effekt für kontinuierliche Wertebereiche nutzen zu können, wird z. B. *Clustering* eingesetzt. Bei der Clusteranalyse werden Algorithmen eingesetzt, die mittels Heuristiken Abschätzungen über zusammenhängende Bereiche des Lösungsraums vornehmen, und aufgrund dessen die Splitting-Strategie bestimmen (vgl. Vu et al. 2003a, b).

Domänen-Splitting ist ein gängiges Verfahren, wenn punktgenaue, kanonische Lösungen benötigt werden. Allerdings mündet das Verfahren bei größeren Constraint-Netzen schnell in einer kombinatorischen Explosion, da u. U. sehr viele Suchzweige berücksichtigt werden müssen (vgl. Lhomme 1993, S. 236 f.). Es bietet sich daher an, Domänen-Splitting mit Filtermechanismen in Form von Konsistenzverfahren zu kombinieren (vgl. Van Hentenryck et al. 1997).[93]

5.3.3 Label Inference

Zur Einschränkung von Intervallgrenzen adaptierte *Ernest Davis* auf intervallarithmetischen Grundlagen den Waltz-Algorithmus zur Filterung ungültiger Werte aus den Domänen der Constraint-Variablen. Davis (1987) nennt den Effekt *Label Inference* und bezeichnet damit das sukzessive Ableiten von Wertebereichseinschränkungen und das Belegen der Variablen bzw. Knoten im Constraint-Netz mit der Menge der jeweils möglichen Werte (engl. *interval labeling*).

[92]Das heißt die Constraints werden berücksichtigt.
[93]Häufig engl. *Branch & Prune* genannt.

procedure REVISE($C(v_1, \ldots, v_k)$):
begin CHANGED $\leftarrow \emptyset$
  for each argument v_i **do**
    begin S $\leftarrow$ REFINE(C, v_i)
      if $S = \emptyset$ **then halt**   /* the original constraints were inconsistent */
      else if $S \neq I_i$ **then**
        begin $I_i \leftarrow S$;
          add v_i to CHANGED
        end
    end
  return CHANGED
end

procedure WALTZ:
begin $Q \leftarrow$ a Queue of all constraints
  while $Q \neq \emptyset$ **do**
    begin remove constraint C from Q;
      CHANGED $\leftarrow$ REVISE(C)
      for each v_i in CHANGED **do**
        for each constraint $C' \neq C$ which has v_i in its domain **do**
          add C' to Q
    end
end

Abbildung 5.23: Intervallpropagation basierend auf dem Waltz-Algorithmus (vgl. Davis 1987, S. 286)

Davis (1987) definiert für den Vorgang des Einschränkens der Intervallgrenzen, auch (engl.) *label refinement* genannt, eine Funktion `REFINE`, die innerhalb der `REVISE`-Prozedur des Waltz-Algorithmus (siehe Abbildung 5.23) aufgerufen wird (vgl. Davis 1987, S. 285):

Definition 5.3.9 (`REFINE`-Funktion)
Sei C ein Constraint mit den Variablen $v_1, \ldots, v_k$. Sei das Intervall I_i der Wertebereich für v_i. Dann sei

$$\mathrm{REFINE}(C, v_j) = \{a_j \in I_j \mid \exists (a_i \in I_i, i = 1, \ldots, k, i \neq j) : C(a_1, \ldots, a_j, \ldots, a_k)\}.$$

Der Aufruf `REFINE`(C, v_j) liefert somit die Menge der Werte für v_j, die konsistent mit dem Constraint C und allen anderen Wertebereichen I_i sind. Ein Wert a_j befindet sich in dieser Ergebnismenge, wenn er in I_j ist und Teil des k-stelligen Tupels $a_1, \ldots, a_k$ ist, der das Constraint C mit den Wertebereichen I_i erfüllt. Eingebettet in die Prozedur `REVISE` schränkt diese nun, sofern dies aufgrund des Constraints C möglich ist, mit Hilfe von `REFINE` der Reihe nach die Wertebereiche für alle beteiligten Variablen $v_1, \ldots, v_k$ ein, und gibt die Menge der Variablen zurück, deren Wertebereiche sich geändert haben. Die Ausführung dieser Version des Waltz-Algorithmus soll an dieser Stelle durch ein Beispiel erläutert werden (vgl. Davis 1987, S. 286 f.):

Beispiel 5.3.5 *Die beiden Constraints*

$$C_{1,2,3} : v_1 + v_2 = v_3 \quad \textit{und} \quad C_{1,2} : v_2 \leq v_1$$

mit den Wertebereichen

$$v_1 : [2, 12], \quad v_2 : [4, 9], \quad v_3 : [2, 8]$$

würden durch den Algorithmus auf die folgende Weise verarbeitet werden: Bei Beginn sind in der Warteschlange Q die beiden Constraints $C_{1,2,3}$ und $C_{1,2}$ enthalten.[94]

$C_{1,2,3}$ wird aus der Warteschlange herausgenommen.
$\rightarrow$ Da $v_1 \geq 2$ und $v_2 \geq 4$, ergibt $C_{1,2,3}$ für $v_3 \geq 6$, daher Einschränkung von $v_3 : [6, 8]$.
$\rightarrow$ Da $v_3 \leq 8$ und $v_2 \geq 4$, ergibt $C_{1,2,3}$ für $v_1 \leq 4$, daher Einschränkung von $v_1 : [2, 4]$.
$\rightarrow$ Da $v_3 \leq 8$ und $v_1 \geq 2$, ergibt $C_{1,2,3}$ für $v_2 \leq 6$, daher Einschränkung von $v_2 : [4, 6]$.
Weil sich die Wertebereiche von v_1 und v_2 geändert haben, wird $C_{1,2}$ zur Warteschlange hinzugefügt.

$C_{1,2}$ wird aus der Warteschlange herausgenommen.
$\rightarrow$ Da $v_1 \leq 4$, ergibt $C_{1,2}$ für $v_2 \leq 4$, daher Einschränkung von $v_2 : [4, 4]$.
$\rightarrow$ Da $v_2 \geq 4$, ergibt $C_{1,2}$ für $v_1 \geq 4$, daher Einschränkung von $v_1 : [4, 4]$.
Weil sich die Wertebereiche von v_1 und v_2 geändert haben, wird $C_{1,2,3}$ zur Warteschlange hinzugefügt.

$C_{1,2,3}$ wird aus der Warteschlange herausgenommen.
$\rightarrow$ Da $v_1 \geq 4$ und $v_2 \geq 4$, ergibt $C_{1,2,3}$ für $v_3 \geq 8$, daher Einschränkung von $v_3 : [8, 8]$.
Weil sich der Wertebereich von v_3 geändert hat, v_3 allerdings nirgends außer in $C_{1,2,3}$ vorkommt, wird kein weiteres Constraint zur Warteschlange hinzugefügt.

Aufgrund der nun leeren Warteschlange wird der Algorithmus beendet. Die Einschränkung der Wertebereiche hat in diesem Fall zu einem eindeutigen Ergebnis in Form von Punktintervallen geführt:

$$v_1 : [4, 4], \quad v_2 : [4, 4], \quad v_3 : [8, 8]$$

Dieser frühe Versuch zur Anwendung von Konsistenztechniken und Constraint-Propagation auf kontinuierliche Domänen war allerdings mit erheblichen Problemen behaftet (vgl. Davis 1987, S. 305; Sam-Haroud 1995, S. 28). So garantiert das eingesetzte Verfahren Konvergenz und Vollständigkeit nur für eingeschränkte Constraint-Klassen. Bei nichtlinearen Constraints ist das Verhalten nicht vorhersagbar und führt (1) häufig zur frühzeitigen Beendigung der Filterung ohne das Konsistenz hergestellt wird (engl. *early quiescence*), bzw. (2) dass der Prozess in eine Endlosschleife läuft (engl. *cycling*) oder (3) nur sehr langsam konvergiert und dadurch zu unakzeptablen Antwortzeiten führt (engl. *slow convergence*).

[94] Nachfolgend werden aus Gründen der Übersichtlichkeit ausschließlich Operationen aufgeführt, die eine Änderung der Wertebereiche bewirken.

Beispiel 5.3.6 *Gegeben sei das einfache ICSP mit den beiden Constraints*

$$C_1 : v_1 = v_2 \qquad\qquad C_2 : v_2 = \frac{v_1}{2}$$

und den Wertebereichen $v_1 : [0, 100]$ *und* $v_2 : [0, 100]$*. Bei jedem Durchlauf des Propagationsalgorithmus werden die Obergrenzen der Intervalle halbiert. Das ICSP ist in dem Augenblick konsistent, wenn* $v_1 = v_2 = 0$*. Dieser Fixpunkt wird allerdings theoretisch nie erreicht, da sich die Wertebereiche ihm nur asymptotisch annähern.*[95]

Davis schlägt mehrere Möglichkeiten vor, dem Problem der Terminierung zu begegnen: Abbruchkriterien könnten z. B. ein Zeitlimit für die Propagation, eine Obergrenze für die Anzahl der Auswertungen einzelner Constraints oder eine Beschränkung der Genauigkeit sein, ab der Intervallgrenzen als identisch angesehen werden (vgl. Davis 1987, S. 305).

Ein weiteres Manko ist, dass die Anwendung des Waltz-Algorithmus lediglich lokale Konsistenz garantiert, d. h. Konsistenz bezogen auf jeweils ein Constraint, und keine globale Konsistenz, bei der die Wertebereiche der Variablen bzgl. sämtlicher Constraints konsistent sind. Trotzdem ist dieses Verfahren für viele Anwendungen bereits ausreichend, und wurde daher in der Vergangenheit in unterschiedlichen Variationen in einer Reihe von Systemen eingesetzt.

5.3.4 Toleranzpropagation

Der Begriff der *Toleranzpropagation* (engl. *tolerance propagation*, TP) wurde von *Eero Hyvönen* eingeführt. Das eigentliche Verfahren ist wiederum eng mit dem Waltz-Filteralgorithmus verwandt, kombiniert allerdings Konsistenztechniken mit speziellen Methoden der Intervallarithmetik zum Erreichen sowohl von lokaler als auch globaler Konsistenz (vgl. Hyvönen 1992). „Toleranzen“ steht bei Hyvönen als Synonym für Intervalle, die wiederum genutzt werden, um unscharfes Wissen modellieren und verarbeiten zu können. Dieses Wissen bezieht sich auf Parameter bzw. Variablen, die sich innerhalb bestimmter Toleranzen befinden, welche durch die Intervallgrenzen der Wertebereiche spezifiziert werden. Eine Situation bzw. Belegung S der Variablen v_i, $i \in \{1, \ldots, n\}$, wird deshalb auch *Toleranzsituation* genannt.

Aufgrund der großen Ähnlichkeit zum Label Inference von Davis (1987) – TP kann genutzt werden, einen ähnlichen Konsistenzgrad herzustellen – besitzt TP dieselbe Problematik bzgl. der Terminierung (vgl. Beispiel 5.3.6). Es ist die Gefahr gegeben, dass der Algorithmus auch bei einfachen Constraints in eine Endlosschleife läuft, wobei Hyvönen (1992, S. 84) hier ebenfalls Präzisionsgrenzen als Gegenmaßnahme vorschlägt, ab denen zwei Intervalle als identisch betrachtet werden.

Der Ansatz von Hyvönen berücksichtigt diskontinuierliche Intervalle, indem eine Intervallzerlegung zur Ausklammerung von Definitionslücken und Beibehaltung der Monotonieeigenschaften bei der Auswertung von impliziten Funktionen erfolgen kann,[96] und stellt

[95] In der Praxis terminiert dieser Vorgang irgendwann, da die Intervallgrenzen im Rechner nur mit begrenzter Genauigkeit repräsentiert werden können.

[96] Ausklammerung undefinierter Bereiche, z. B. $\frac{1}{0}$.

einen entsprechend erweiterten Algorithmus zur Verfügung, der die einzelnen Teilintervalle einer Zerlegung berücksichtigt (vgl. Hyvönen 1992, S. 93). Dadurch wird sichergestellt, dass als Ergebnisintervalle von einzelnen Rechenoperationen jeweils ununterbrochene Intervalle vorliegen. Allerdings wird damit auch vermehrter Aufwand sowohl durch die Zerlegung der Intervalle als auch bei der Berechnung und Berücksichtigung der auftretenden, unterschiedlichen Toleranzsituationen in Kauf genommen, die vereinigt jeweils die (diskontinuierlichen) Lösungsintervalle der Constraint-Variablen bilden. Hyvönen (1992, S. 79) weist auch auf die daraus resultierenden Probleme bzgl. der entstehenden Komplexität hin und verweist ebenfalls auf die alternative Möglichkeit, trotz der entstehenden Ungenauigkeiten, ausschließlich ununterbrochene Intervalle zu verwenden (vgl. Abschnitt 5.3.1.4, S. 149).

5.3.4.1 Lokale Toleranzpropagation

Ähnlich der Kantenkonsistenz für finite Domänen definiert Hyvönen einen lokalen Konsistenzgrad für die TP (vgl. Hyvönen 1992, S. 77):

Definition 5.3.10 (lokal konsistente Toleranzsituation)
Eine Toleranzsituation ist lokal konsistent, gdw. alle Variablen v_i, $i \in \{1, \ldots, n\}$ lokal konsistent sind, d. h. für alle Constraints C_j, $j \in \{1, \ldots, m\}$ gilt, dass für jede in C_j vorkommenden Variablen v_i zu jeder zulässigen Belegung $v_i = x$, $x \in I_i$ jeweils eine gültige Belegung der übrigen in C_j vorkommenden Variablen existiert, so dass C_j erfüllt ist.

Während Davis (1987) eine spezielle Funktion `REFINE` definiert, welche die Constraints im Ganzen propagiert und daher in dieser Form, abhängig von der Implementierung, jeweils nur bestimmte Constraint-Arten verarbeiten kann, basiert die TP auf dem Konzept der *solution functions*. Dies beruht darauf, dass jede Gleichung eine Menge von Funktionen impliziert. Ein Constraint C_j, welches die Variablen $v_{j_1}, \ldots, v_{j_k}$ mit $k \leq n$ enthält, ergibt die folgenden Funktionen:

$$\begin{aligned} v_{j_1} &= F_{j_1}(v_{j_2}, \ldots, v_{j_k}) \\ v_{j_2} &= F_{j_2}(v_{j_1}, v_{j_3}, \ldots, v_{j_k}) \\ &\vdots \\ v_{j_k} &= F_{j_k}(v_{j_1}, \ldots, v_{j_{(k-1)}}) \end{aligned}$$

Beispiel 5.3.7 *Die impliziten Funktionen bzw. „solution functions" für das einfache Additions-Constraint $C_{1,2,3} : v_1 + v_2 = v_3$ lauten:*

$$\begin{aligned} v_1 = F_1(v_2, v_3) = v_3 - v_2 \\ v_2 = F_2(v_1, v_3) = v_3 - v_1 \\ v_3 = F_3(v_1, v_2) = v_1 + v_2 \end{aligned}$$

Ein k-stelliges Constraint $C_j(v_{j_1}, \ldots, v_{j_k})$ besitzt k implizite Funktionen, je eine für jede Variable v_{j_i}, $i \in \{1, \ldots, k\}$. Die Konsistenzbedingungen von C_j sind dann erfüllt, wenn alle impliziten Funktionen erfüllt sind. Anstatt der ursprünglichen Constraints werden daher bei der TP die *solution functions* ausgewertet und propagiert (vgl. Hyvönen 1989, S. 1195). Da es bei komplexeren Gleichungen schwierig und teilweise unmöglich ist,[97] implizite Funktionen anzugeben, nimmt Hyvönen zunächst eine Vereinfachung des Gleichungssystems durch Zerlegung der Constraints in primitive Constraint-Ausdrücke vor. Teile von komplexen Ausdrücken werden dabei durch Einführung neuer Variablen in zusätzliche Gleichungen überführt. Diese Substitution führt dazu, dass implizite Funktionen leicht erkennbar werden.

Beispiel 5.3.8 *Das ICSP mit dem Constraint*

$$C_1 : v_1^3 + v_2 \times v_3 = -v_4$$

ist äquivalent zu dem vereinfachten ICSP bestehend aus den folgenden primitiven Constraints:

$$\begin{array}{ll} C_1 : v_5 = v_1^3 & \qquad C_3 : v_7 = -v_4 \\ C_2 : v_6 = v_2 \times v_3 & \qquad C_4 : v_7 = v_5 + v_6 \end{array}$$

Durch die Einführung neuer Variablen und Termsubstitutionen müssen bei der Propagation ausschließlich primitive Constraint-Ausdrücke behandelt werden, die maximal drei Variablen beinhalten. Dies erleichtert wesentlich die Erstellung der zur Propagation benötigten *solution functions*.

Hyvönen (1992) bietet durch die lokale TP mit intervallwertigen *solution functions* einen flexibleren Ansatz als Davis (1987), ist dem Waltz-Filteralgorithmus aber trotzdem sehr ähnlich: Lokale TP stellt lokale Konsistenz her, d. h. Korrektheit ist gewährleistet, es gehen keine vorhandenen Lösungen verloren (wenn es welche gibt, befinden sie sich in den Lösungsintervallen). Allerdings ist das Verfahren ebenfalls unvollständig, da nicht garantiert ist, dass es Lösungen gibt, auch wenn die lokale TP keine (lokale) Inkonsistenz feststellen konnte. Lediglich wenn mittels lokaler TP eine Inkonsistenz aufgefunden wird, kann mit Bestimmtheit gesagt werden, dass keine (globale) Lösung existiert.

5.3.4.2 Globale Konsistenz durch Toleranzpropagation

Analog zu dem Begriff der globalen Konsistenz (vgl. Abschnitt 5.2.3.5, S. 104) unterscheidet Hyvönen zwischen lokal und global konsistenten Toleranzsituationen (vgl. Hyvönen 1992, S. 77):

Definition 5.3.11 (global konsistente Toleranzsituation)
Eine Toleranzsituation ist global konsistent, gdw. alle Variablen v_i, $i \in \{1, \ldots, n\}$ global konsistent sind, d. h. es existiert zu jeder beliebigen zulässigen Belegung $v_i = x$, $x \in I_i$ jeweils eine gültige Belegung aller übrigen Variablen, so dass sämtliche Constraints C_j, $j \in \{1, \ldots, m\}$ erfüllt sind.

[97] Beispielsweise $x + \log(x) = 0$.

Um sicherzustellen, dass es Lösungen für ein bestimmtes Problem gibt, ist globale Konsistenz erforderlich. Aus Definition 5.3.10 auf Seite 155 und Definition 5.3.11 auf der vorherigen Seite ist ersichtlich, dass durch globale Konsistenz lokale Konsistenz impliziert wird.

Azyklische Constraint-Netze Umgedreht allerdings impliziert lokale Konsistenz die globale Konsistenz nur selten, nämlich immer dann, wenn ein lokal konsistentes Constraint-Netz keine Zyklen enthält (vgl. Hyvönen 1991, S. 246; Hyvönen 1992, S. 89 f.):

Definition 5.3.12 (globale Konsistenz in azyklischen Constraint-Netzen)
Ein azyklisches Constraint-Netz ist global konsistent, gdw. es lokal konsistent ist.

Wenn für eine Variable v_i, $i \in \{1, \ldots, n\}$ eine zulässige Belegung $v_i = x$, $x \in I_i$ gewählt wird, so lassen sich wegen der lokalen Konsistenz den jeweils restlichen Variablen, die mit v_i in einem Constraint vorkommen, ebenfalls konsistente Werte zuweisen. Wegen der Zyklenfreiheit führt nun jede Kante aus dem bisher betrachteten Graphen in einen eigenständigen Teilgraphen, der auf dieselbe Art mit Werten belegt werden kann. Dies lässt sich mit jeder beliebigen Variable v_i und jedem Wert x aus deren Wertebereich durchführen, so dass in diesem Fall die lokale Konsistenz mit der globalen Konsistenz äquivalent ist.

Dies geht wiederum direkt auf die von Freuder (1982) gemachten Beobachtungen zurück (vgl. Abschnitt 5.2.3.5, S. 105). Wenn auch globale Konsistenz durch lokale TP nur selten erreicht wird, so kann dennoch versucht werden, wenigstens engere Intervallgrenzen zu bestimmen. Dies kann durch *dynamic splitting* erreicht werden. Dabei wird eine lokal konsistente Toleranzsituation bzgl. der global inkonsistenten Variablen durch Zerteilen der Wertebereiche eben dieser Variablen in unterschiedliche Teilprobleme zerlegt (vgl. Abschnitt 5.3.2, S. 150). Von diesen Teilproblemen werden wiederum die lokalen Lösungen bestimmt usw. Dies geschieht so lange, bis keine Änderungen mehr eintreten bzw. u. U. globale Konsistenz erreicht wird. Die Variablen, die für diese Zerlegung ausgewählt werden, sind i. d. R. die sog. *cutset*-Variablen. Ein *cutset* besteht dabei aus der kleinsten Menge der Variablen, durch die sämtliche Zyklen im Constraint-Netz verbunden werden.[98] Das *dynamic splitting* funktioniert deswegen, weil durch die beschränkteren Wertebereiche der Variablen globale Lösungen einfacher anhand lokaler Kriterien hergestellt werden können (vgl. Hyvönen 1991, S. 247). Führt *dynamic splitting* sogar dazu, dass die *cutset*-Variablen jeweils auf einen einzigen Wert, d. h. ein Punktintervall, reduziert werden können, so ist das Constraint-Netz durch die lokale Konsistenz gleichzeitig global konsistent: Die *cutset*-Variablen können in diesem Fall als Konstanten betrachtet werden. Das resultierende Constraint-Netz ist azyklisch (vgl. Hyvönen 1992, S. 99 ff.).[99]

[98] „[A 'cutset' is] a minimal set of nodes, that 'cuts' every cycle in a net“ (Hyvönen 1992, S. 99).

[99] Diese Strategie kann auch sinnvoll zum Auffinden von Lösungen in CSPs mit finiten Domänen eingesetzt werden, und ist dort unter den Namen *Cutset Conditioning* und *Cycle-Cutset*-Methode bekannt (vgl. Dechter 1990a; Russel und Norvig 2002): Wenn ein *cutset* identifiziert und mit konsistenten Werten belegt wurde, lassen sich, falls eine Lösung existiert, die Variablen im verbleibenden (baumartigen) Constraint-Graphen relativ einfach mit konsistenten Werten belegen (vgl. Dechter und Pearl 1987, 1989). Wegen der

Globale Toleranzpropagation Neben dem Ansatz des *dynamic splitting*, der bereits unter geeigneten Umständen zu globaler Konsistenz führt, gibt es ein weiteres Verfahren mit dem sich global konsistente Toleranzsituationen herstellen lassen. Ermöglicht wird dies, weil in der TP die Konsistenzbedingungen als *solution functions* angegeben werden (vgl. Hyvönen 1992, S. 83). Da eine lokal konsistente Toleranzsituation nur dann gleichzeitig global konsistent ist, wenn das Constraint-Netz zyklenfrei ist, lässt sich globale Konsistenz erreichen, indem für die Constraints bzgl. kritischer Variablen algebraische Gleichungsumformungen zur Eliminierung eben dieser Variablen vorgenommen werden. Die durch diese algebraischen Transformationen entstehenden Konsistenzbedingungen werden *global solution functions* genannt (vgl. Hyvönen 1992, S. 101 ff.):[100]

Beispiel 5.3.9 *Für das ICSP mit den beiden Constraints*

$$C_1 : v_1 + v_2 = v_3 \qquad\qquad C_2 : v_2 + v_3 = v_4$$

liefert lokale TP keine global konsistente Lösung. Der Constraint-Graph enthält einen Zyklus, da sowohl v_2 als auch v_3 in beiden Constraints enthalten ist. Wenn zur Eliminierung dieser Variablen die Gleichung des Constraints C_2 jeweils nach v_2 und v_3 aufgelöst in die Gleichung von C_1 eingesetzt wird, erhält man:

$$v_1 + v_2 = (v_4 - v_2) \qquad\qquad v_1 + (v_4 - v_3) = v_3$$

Aufgelöst nach v_2 und v_3 ergibt dies die beiden folgenden neuen global solution functions:

$$v_2 = \frac{v_4 - v_1}{2} \qquad\qquad v_3 = \frac{v_1 + v_4}{2}$$

Zusammen mit den beiden solution functions für v_1 und v_4 ergibt sich die folgende Agenda auszuwertender Funktionen für den Propagationsvorgang:

$$v_1 = v_3 - v_2, \qquad v_2 = \frac{v_4 - v_1}{2}, \qquad v_3 = \frac{v_1 + v_4}{2}, \qquad v_4 = v_2 + v_3$$

Die restlichen lokalen solution functions ($v_2 = v_3 - v_1$), ($v_3 = v_1 + v_2$), ($v_2 = v_4 - v_3$) und ($v_3 = v_4 - v_2$) finden keine Anwendung, da sie für den ursprünglichen Zyklus im Constraint-Netz verantwortlich sind.

Die resultierenden impliziten Funktionen erlauben bei Anwendung von lokaler TP die Herstellung einer global konsistenten Toleranzsituation. Dieses Vorgehen wird *globale Toleranzpropagation* genannt. Problematisch ist die globale TP, weil nicht alle Gleichungen algebraisch umformbar sind, und universelle Verfahren bis heute nicht bekannt sind. In der Praxis sind diese Umformungen häufig so komplex, dass die globale Konsistenz als nur theoretisch erreichbar angesehen werden muss. Im praktischen Einsatz ist lokale TP

---

großen Abhängigkeit der Effektivität von der Struktur des jeweiligen Constraint-Graphen, und der starken heuristischen Komponente dieses Verfahrens bei der Auswahl der *cutset*-Variablen, wurde es in dieser Arbeit nicht nicht näher betrachtet (vgl. Abschnitt 5.2.2, S. 89).

[100] Auch: *universal solution functions* und *universal relations* (vgl. Hyvönen 1989, S. 1197).

anwendbar, wobei Hyvönen darauf hinweist, dass Constraint-Solver inkrementell erweitert werden können, so dass sie nach und nach eine größere Menge Transformationsregeln oder *dynamic splitting* unterstützen (vgl. Hyvönen 1991, S. 249). Auf diese Weise werden möglichst viele Fälle abgedeckt, in denen globale Konsistenz herstellbar ist. Für alle weiteren Fälle werden höhere Grade lokaler Konsistenz erreichbar, auch wenn globale Konsistenz nicht garantiert werden kann.

5.3.5 2B-, 3B- und *k*B-Konsistenz

Erweiterungen zur Intervallpropagation wurden von *Olivier Lhomme* eingeführt. Neben einem mit Kantenkonsistenz vergleichbaren lokalen Konsistenzgrad mit dem Namen *arc B-consistency*, der vereinfacht in neuerer Literatur ausschließlich *2B-consistency* bzw. *2B-Konsistenz* genannt wird, definiert Lhomme (1993) den höheren lokalen Konsistenzgrad *3B-Konsistenz* und erweitert den Konsistenzbegriff der Intervallpropagation auf *kB-Konsistenz*.[101] Lhomme (1993) berücksichtigt ausschließlich ununterbrochene Intervalle. Der Gedanke ist, auch in Bezug auf diskrete Domänen, dass, indem nur die Intervallgrenzen (engl. *bounds*) betrachtet werden, durch diese Art der Komplexitätsreduktion eine kombinatorische Explosion vermieden wird, wie sie bei umfangreichen (unterbrochenen) Domänen und komplexen Constraints auftreten kann. Weil diese Art Konsistenz lediglich die konvexe äußere Hülle der Intervalle betrachtet, ist 2B-Konsistenz ebenfalls unter dem Namen *hull consistency* bzw. *Hull-Konsistenz* bekannt (vgl. Benhamou 1995, 2001; Benhamou et al. 1999a; Collavizza et al. 1998, 1999).

5.3.5.1 Hull-Konsistenz

Die 2B- bzw. Hull-Konsistenz ist vergleichbar mit der lokalen Konsistenz, die beim Label Inference von Davis (1987) bzw. bei der lokalen Toleranzpropagation von Hyvönen (1992) bzgl. ununterbrochener Intervalle hergestellt wird. Lhomme (1993, S. 235) definiert 2B-Konsistenz folgendermaßen:

Definition 5.3.13 (2B-Konsistenz/arc B-consistency)
Sei P ein ICSP, v_i*,* $i \in \{1, \ldots, n\}$ *eine Variable von P,* $I_i = [a, b]$ *und* C_j*,* $j \in \{1, \ldots, m\}$ *ein Constraint in P.* I_i *ist 2B-konsistent, gdw.* $\forall C_j(v_i, v_1, \ldots, v_k)$ *ein Constraint über* v_i

$\exists v_1, \ldots, v_k \in I_1 \times \ldots \times I_k \mid C_j(a, v_1, \ldots, v_k)$ *ist erfüllt,*

$\exists v_1, \ldots, v_k \in I_1 \times \ldots \times I_k \mid C_j(b, v_1, \ldots, v_k)$ *ist erfüllt.*

Ein ICSP ist 2B-konsistent, gdw. alle Wertebereiche I_i *2B-konsistent sind.*

Ein Constraint C_j ist demnach 2B-konsistent, wenn es für jede Variable v_i für alle anderen Variablen aus C_j mögliche Belegungen gibt, die C_j erfüllen, wenn v_i jeweils mit seiner oberen und unteren Grenze belegt wird. Dies entspricht einer auf die Intervallgrenzen der Wertebereiche bezogenen Form von Kantenkonsistenz.

Um die Wertebereichseinschränkungen, eingebettet in eine Art Waltz-Filteralgorithmus, vornehmen zu können, müssen wiederum Funktionen für jede Variable eines Constraints abgeleitet werden. Lhomme (1993, S. 234) nennt dies „Projektion“:

[101] „B“ jeweils für *bound*, von engl. *bound propagation*.

Definition 5.3.14 (Projektion eines Constraints)
Seien $(v_1, \ldots, v_k)$ k Variablen, sei die Box $B^\Box = I_1 \times \cdots \times I_k$. Die Projektion über v_i des Constraints $C_j(v_1, \ldots, v_k)$ bzgl. der Domänen $I_1, \ldots, I_k$, die mit $\Pi_i(C_j(v_1, \ldots, v_k), B)$ benannt wird, ist die Menge: $\{x_i \in I_i \mid \exists (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k) \in I_i \times \cdots \times I_{i-1} \times I_{i+1} \times \cdots \times I_k$, so dass $C_j(x_1, \ldots, x_k)$ erfüllt wird$\}$.

Der Wertebereich einer Variablen v_i wird demnach eingeschränkt durch die Berechnung einer Projektion Π_i von einem Constraint C_j über die Variable v_i im Lösungsraum, der von den in C_j vorkommenden Variablen außer v_i aufgespannt wird. Diese Projektionen entsprechen den impliziten *solution functions* von Hyvönen (vgl. Abschnitt 5.3.4.1, S. 155).[102] Sie liefern bzgl. einzelner Constraints konsistente Ergebnismengen für die jeweiligen Variablen. Um diese Berechnungen vornehmen zu können, nimmt Lhomme (1993) ebenso wie Hyvönen (1992) eine Zerlegung komplexer Ausdrücke in ternäre *basic constraints* vor.[103] Für diese *basic constraints* lassen sich Projektionen auf triviale Art und Weise erstellen.

5.3.5.2 Höhere Konsistenzgrade

Da das Konzept von Lhomme ausschließlich die konvexen Intervallgrenzen berücksichtigt, ist 2B-Konsistenz grundsätzlich schwächer als Kantenkonsistenz. Wenn ein Constraint für eine Variable zu einem diskontinuierlichen Lösungsintervall führt, werden ausschließlich die obere und untere Grenze des Intervalls berücksichtigt, was lokale Inkonsistenzen innerhalb der Intervallgrenzen erlaubt. Nur für konvexe Constraints, die nicht zu diskontinuierlichen Ergebnisintervallen führen (engl. *non-disjunctive* Constraints), ist 2B-Konsistenz äquivalent zur Kantenkonsistenz. Höhere Konsistenzgrade lassen sich mittels 3B- und kB-Konsistenz erreichen. Ähnlich wie Kantenkonsistenz zur Pfadkonsistenz verallgemeinert wird, kann 3B-Konsistenz als eine Verallgemeinerung von 2B-Konsistenz gesehen werden. Entsprechend gilt dies für k-Konsistenz und kB-Konsistenz.

Zur Definition von 3B-Konsistenz ist die Einführung der folgenden Notation notwendig: Für jedes CSP gibt es einen Punkt Φ, an dem der Filtervorgang abgeschlossen ist, wenn eine bestimmte Konsistenz λ erreicht ist (engl. *closure*). Dieser konsistente Zustand Φ_λ ist für jede Konsistenzart und jedes CSP einzigartig. Der konsistente Zustand eines ICSP P für 2B-Konsistenz wird $\Phi_{2B}(P)$ genannt. Weiterhin wird die Vereinigung von einem ICSP $P = (V, I, C)$ und einem Constraint k (welches ausschließlich in P enthaltene Variablen beschränkt) definiert durch: $P' = P \cup \{K\} = (V, I, C \cup \{K\})$. Die Definition von 3B-Konsistenz nach Lhomme (1993, S. 237) lautet demnach folgendermaßen:

Definition 5.3.15 (3B-Konsistenz/3B-consistency)
Sei P ein ICSP, v_i, $i \in \{1, \ldots, n\}$ eine Variable von P und $I_i = [a, b]$. I_i ist 3B-konsistent, gdw. P' und P'' beide nicht leer sind, wobei
$P' = \Phi_{2B}(P \cup \{v_i = a\})$ *und* $P'' = \Phi_{2B}(P \cup \{v_i = b\})$.
Ein ICSP ist 3B-konsistent, gdw. alle Wertebereiche I_i 3B-konsistent sind.

[102] Von Lhomme et al. (1998, 1996) auch *narrowing functions* genannt, von *to narrow* (engl.): eingrenzen, schmälern, verengen.

[103] Bei Hyvönen „primitive“ Constraints genannt.

Verallgemeinert ausgedrückt wird 3B-Konsistenz erreicht, indem überprüft wird, ob 2B-Konsistenz für das gesamte ICSP hergestellt werden kann, wenn der Wertebereich einer Variable jeweils auf die untere und auf die obere Intervallgrenze gesetzt ist (vgl. Collavizza et al. 1999, S. 214).[104] Der Zustand von erreichter 3B-Konsistenz für das ICSP P wird $\Phi_{3B}(P)$ genannt.

Analog kann der Konsistenzbegriff auf die kB-Konsistenz verallgemeinert werden, um entsprechend höhere Grade lokaler Konsistenz bezogen auf die Intervallgrenzen zu erreichen (vgl. Lhomme 1993, S. 238). Die Definition erfolgt rekursiv: So wie 3B-Konsistenz die 2B-Konsistenz nutzt, kann 3B-Konsistenz eingesetzt werden, um 4B-Konsistenz zu erreichen usw. (vgl. Bordeaux et al. 2001, S. 305; Lebbah und Lhomme 2002, S. 119).

5.3.5.3 Erweiterungen von Hull-Konsistenz

Um dem Problem der Terminierung des Propagationsvorgangs bei zyklischen Constraints zu begegnen, schlägt Lhomme (1993) zur Präzisionsbegrenzung eine partielle Form von 2B-Konsistenz, die *2B(w)-Konsistenz*, vor und führt einen entsprechend modifizierten Algorithmus ein. Mit 2B(w)-Konsistenz wird der Grad lokaler Konsistenz gekennzeichnet, der erreicht wird, wenn die Propagation vor Erreichen von 2B-Konsistenz, bzw. der Präzision der Fließkomma-Darstellung des eingesetzten Rechnersystems, vorzeitig terminiert. Durch w (von engl. *width*) wird die erlaubte Ungenauigkeit der berechneten Intervallgrenzen charakterisiert, wobei für $w = 0$ die 2B(w)-Konsistenz äquivalent mit 2B-Konsistenz ist. Analog existiert für 3B-Konsistenz eine Erweiterung *3B*(w_1, w_2)*-Konsistenz*, wobei w_1 für die Präzision der Intervallgrenzen von 3B-, und w_2 für die Präzision der eingesetzten 2B-Konsistenz steht. Wiederum ist 3B$(0, 0)$-Konsistenz äquivalent mit 3B-Konsistenz.

Weitere Optimierungen bzgl. des *cycling* und *slow convergence* (vgl. Abschnitt 5.3.3, S. 153) und verbesserte Algorithmen der vorgestellten Verfahren finden sich in den Arbeiten von Lhomme et al. (1998, 1996) und Lebbah und Lhomme (1998, 2002). So besteht ein Ansatz darin, nur relevante Projektionen bzw. *solution functions* pro Variable einzusetzen, bzw. deren Reihenfolge geschickt zu verändern, so dass weniger wichtige und zyklenauslösende Funktionen erst dann berechnet werden, wenn bereits ein Fixpunkt erreicht ist (vgl. Lhomme et al. 1998, 1996). Eine weitere Möglichkeit besteht darin, mittels mathematischer Grenzwertberechnungen in bestimmten Situationen eine Hochrechnung über die voraussichtliche Konvergenz der Intervallgrenzen und eine entsprechende Beschleunigung dieses Vorgangs vorzunehmen (vgl. Lebbah und Lhomme 1998, 2002).[105]

Frédéric Benhamou et al. stellen einen neuen Algorithmus `HC-4` vor, der 2B- bzw., Hull-Konsistenz ohne eine vorherige Zerlegung in *basic constraints* bzw. primitive Con-

[104] Ähnlich der *singleton consistency* (vgl. Abschnitt 5.2.3.6, S. 110).

[105] Algorithmen zur Herstellung von 3B-Konsistenz prüfen mit Hilfe von 2B-Konsistenz, dass in einem bestimmten Teil des Lösungsraums keine mögliche Lösung enthalten ist, und löschen diesen Teil ggf., indem die entsprechenden Intervallgrenzen eingeschränkt werden. Wenn frühzeitig festgestellt werden kann, dass eine Hochrechnung gegen eine 2B-konsistente Grenze konvergiert (was sich recht einfach gestaltet), ist es sinnvoll, diese nicht separat auf 2B-Inkonsistenz zu testen, und dadurch insgesamt das Verfahren stark zu beschleunigen (vgl. Lebbah und Lhomme 2002, S. 127).

straints herstellt (vgl. Benhamou et al. 1999a).[106] Der Algorithmus leistet dies, indem der Baum der Repräsentation eines jeden einzelnen Constraint-Ausdrucks sowohl in Vorwärts- als auch Rückwärtsrichtung durchlaufen wird, und dabei jeweils die Wertebereiche der Variablen an die propagierten Grenzen angepasst werden. Von Vorteil ist der verringerte Berechnungsaufwand und Speicherbedarf, da primitive Constraints nicht generiert und insgesamt weniger Constraints propagiert werden müssen (vgl. Benhamou et al. 1999a, S. 235).

Boi V. Faltings zeigt wie höhere lokale Konsistenz erreicht werden kann, indem jeweils alle Constraints, welche dieselben Variablen betreffen, zu totalen Constraints zusammengefasst werden (vgl. Definition 4.1.2, S. 51). Lokale Konsistenz auf totalen Constraints ermöglicht die Berechnung engerer Intervallgrenzen, setzt allerdings eine komplexe Auswertung der Constraints einschließlich Extremwertberechnungen voraus (vgl. Faltings 1994). Das vorgestellte Verfahren arbeitet ausschließlich auf binären Constraints, eine Generalisierung ist möglich, erfordert jedoch anspruchsvolle mathematische Vorverarbeitung der Constraints (vgl. Faltings und Gelle 1997).

Ebenfalls höhere Konsistenz garantiert die *global hull consistency*, erstmals erwähnt in der Arbeit von Cruz und Barahona (1999). Globale Hull-Konsistenz wird erreicht, indem überprüft wird, ob globale (kanonische) Lösungen möglich sind, wenn jeweils eine Variable auf ihre Intervallgrenzen gesetzt wird. Anders als bei Varianten der kB-Konsistenz, die das restliche ICSP jeweils lediglich auf (k-1)B-Konsistenz überprüfen, wird hier auf globale Konsistenz der Intervallgrenzen getestet. Eine vollständige Beschreibung sowie entsprechende Algorithmen werden von Cruz und Barahona (2001, 2003) vorgestellt. Globale Hull-Konsistenz wird insbesondere zur Lösung von System mit Differentialgleichungen eingesetzt. Effiziente Algorithmen werden unterstützt von lokalen Suchverfahren. Aufgrund der hohen Komplexität ist globale Hull-Konsistenz derzeit jedoch bei einer größeren Anzahl Variablen nicht effizient anwendbar.

5.3.6 Box-Konsistenz

Sind die bisher vorgestellten Lösungsmethoden für ICSPs jeweils Abwandlungen bzw. Erweiterungen des Waltz-Filteralgorithmus, stellen *Frédéric Benhamou* et al. ein Konsistenzverfahren vor, welches ein numerisch-mathematisches Verfahren zum Einschränken der Wertebereiche von Constraint-Variablen nutzt (vgl. Benhamou et al. 1994a, b). Der erreichte Grad lokaler Konsistenz wird *Box-Konsistenz* genannt (engl. *box consistency*) und stellt eine Vereinigung zwischen Lösungsmethoden für CSPs aus der *AI-Community* und dem Bereich der Intervallanalysis dar. Box-Konsistenz ist ebenfalls eine Approximation von Kantenkonsistenz, die in kontinuierlichen Domänen, aufgrund der Präzisionsbeschränkungen von Rechenanlagen, grundsätzlich nicht erreicht werden kann. Benhamou et al. (1994a) setzen die Strategie des `AC-3`-Algorithmus und das Newton-Intervallverfahren zur Wertebereichseinschränkung ein. Das Newton-Intervallverfahren ist eine Erweiterung des iterativen Newton-Verfahrens zur numerischen Nullstellenberechnung. In Verbindung mit einer internen Splitting-Technik ist es möglich, durch die Intervall-Version dieses Verfah-

[106] Der ursprüngliche Algorithmus von Lhomme (1993) wird hier in Anlehnung an `AC-3` mit `HC-3` benannt (vgl. Benhamou 1995; Benhamou und Older 1997).

rens Box-Konsistenz für nichtlineare Gleichungen und Ungleichungen herzustellen indem die jeweils äußerste linke und rechte Nullstelle einer Intervallfunktion berechnet wird.

Ebenso wie Hull-Konsistenz und deren verwandten Konsistenzgrade beschränkt sich Box-Konsistenz auf konvexe Domänen, d. h. ununterbrochene Intervalle. Der Fokus liegt hier darauf, die Außengrenzen dieser Wertebereiche so weit wie möglich einzuschränken. Die Definition von Box-Konsistenz ist dementsprechend allgemein gehalten und wird in neueren Publikationen übersichtlich folgendermaßen notiert (vgl. Collavizza et al. 1999, S. 220):

Definition 5.3.16 (Box-Konsistenz/box consistency)
Sei P ein ICSP, C_j, $j \in \{1, \ldots, m\}$ ein k-stelliges Constraint in P über die Variablen $(v_1, \ldots, v_k)$. C_j ist Box-konsistent, gdw. $\forall v_i \in \{v_1, \ldots, v_k\}$ mit $I_i = [a, b]$ die folgenden Relationen gelten:

1. $C_j(I_1, \ldots, I_{i-1}, [a, a^+[, I_{i+1}, \ldots, I_k)$,
2. $C_j(I_1, \ldots, I_{i-1},]b^-, b], I_{i+1}, \ldots, I_k)$.

Ein ICSP ist Box-konsistent, gdw. alle Constraints C_j Box-konsistent sind.

Wobei a^+ für die kleinste innerhalb der Präzisionsgrenzen darstellbare Zahl größer als a, und a^- für die größte darstellbare Zahl kleiner als a steht. Durch die Definition wird lediglich bezeichnet, dass jeweils das i-te Intervall nicht weiter eingeschränkt werden kann. Der Zustand erreichter Box-Konsistenz für das ICSP P wird $\Phi_{Box}(P)$ genannt.

Damit die Constraints durch ein numerisches Näherungsverfahren verarbeitet werden können, müssen sie sich in der Form $E \diamond 0$ befinden, wobei E ein Term ist, und $\diamond \in \{=, \geq\}$.[107] Ein wichtiger Aspekt bei der Herstellung von Box-Konsistenz ist die Art und Weise, wie die Generierung von Projektionen erfolgt. Diese werden, wie in den vorher bereits beschriebenen Verfahren, dazu benötigt, den Wertebereich einzelner Variablen einzuschränken. Ein Constraint ist entsprechend Box-konsistent, wenn alle seine Projektionen Box-konsistent sind. Benhamou et al. (1994a) erzeugen Projektionen, indem sie in den ursprünglichen Constraints jeweils sämtliche Variablen bis auf eine durch ihre Intervalldomänen ersetzen. Diese Projektions-Constraints bilden ein System von unären Intervallfunktionen, die von numerischen Verfahren behandelt werden können. Eine Zerlegung komplexer Constraints in primitive Constraints und die Einführung zusätzlicher Variablen, wie bei der Hull-Konsistenz, entfällt, was sich positiv auf die Effizienz bei der Lösung größerer Probleme auswirkt. Zudem sind die Ergebnisse i. A. genauer als die mit 2B-Konsistenz generierten Lösungen. Allerdings sind die Voraussetzungen zur Herstellung von Box-Konsistenz durch numerische Verfahren strenger und die Ergebnisse weniger präzise als 3B-konsistente Lösungen (vgl. Collavizza et al. 1998, S. 158; Collavizza et al. 1999, S. 224; Sam-Haroud 1995, S. 31).

5.3.6.1 Newton-Intervallverfahren

Das Newtonsche (Näherungs-)Verfahren aus der Numerik dient zur numerischen Bestimmung von Nullstellen nichtlinearer Gleichungen. Die Aufgabe, eine reelle Gleichung mit

[107] Die Umformung gestaltet sich trivial, indem jeweils die rechte Seite von der linken Seite einer Gleichung bzw. Ungleichung subtrahiert und der resultierende Term gleich 0 gesetzt wird.

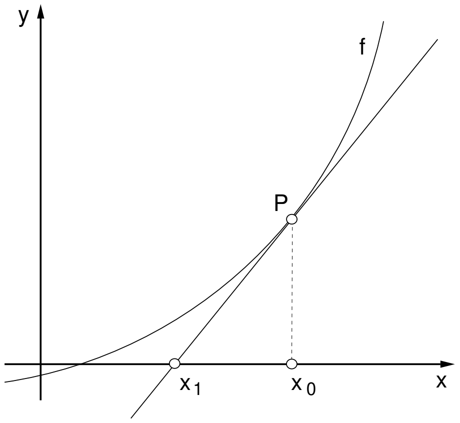


Abbildung 5.24: Newton-Näherungsverfahren zur Nullstellenberechnung (vgl. Embacher und Oberhuemer 2003)

einer Unbekannten zu lösen, ist äquivalent mit dem Problem, die Nullstellen einer Funktion zu berechnen. Solch eine Gleichung kann deshalb immer in der Form $f(x) = 0$ geschrieben werden. Wenn die Funktion f differenzierbar ist, gestattet es das Newton-Verfahren in vielen Fällen, die Nullstellen sehr schnell und mit hoher Präzision zu ermitteln (vgl. Embacher und Oberhuemer 2003).

Angefangen mit einem Schätzwert x_0 für die zu berechnende Nullstelle, werden iterativ weitere Näherungswerte $x_1, x_2, \ldots$ durch Einsetzen in die folgende, rekursive Formel berechnet (vgl. Bronstein et al. 1996, S. 1137):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Die grafische Interpretation des Verfahrens ist in Abbildung 5.24 zu sehen. Beginnend mit dem ersten Schätzwert x_0 wird eine Tangente durch den Punkt P an den Graphen von f gelegt. Die Tangente schneidet die x-Achse in x_1, was in diesem Fall bereits eine deutliche Verbesserung darstellt. Für x_2 wird dasselbe Verfahren auf x_1 angewendet usw. Bereits nach wenigen Iterationen liegen häufig sehr genaue Näherungswerte vor.

Dieses Verfahren kann für die Intervallrechnung erweitert werden, wie von Krawczyk (1969, S. 192 ff.) bzw. Moore (1969, S. 84 ff.) dargelegt wird. Stark vereinfacht geschieht dies, indem für ein Intervall $X = [a, b]$ und $m(X) =$ der Mittelpunkt von X, die Intervallfunktion N (für „Newton") definiert wird:

$$N(X) = m(X) - \frac{F(m(X))}{F'(X)}$$

Beginnend mit einem Ausgangsintervall X_0 wird durch

$$X_{n+1} = N(X_n) \cap X_n$$

die konvergierende Intervallfolge $X_1, X_2, \ldots$ berechnet. Bei jedem Iterationsschritt verkleinert sich dieser Bereich, wodurch, wie bei der reellen Version des Newton-Verfahrens,

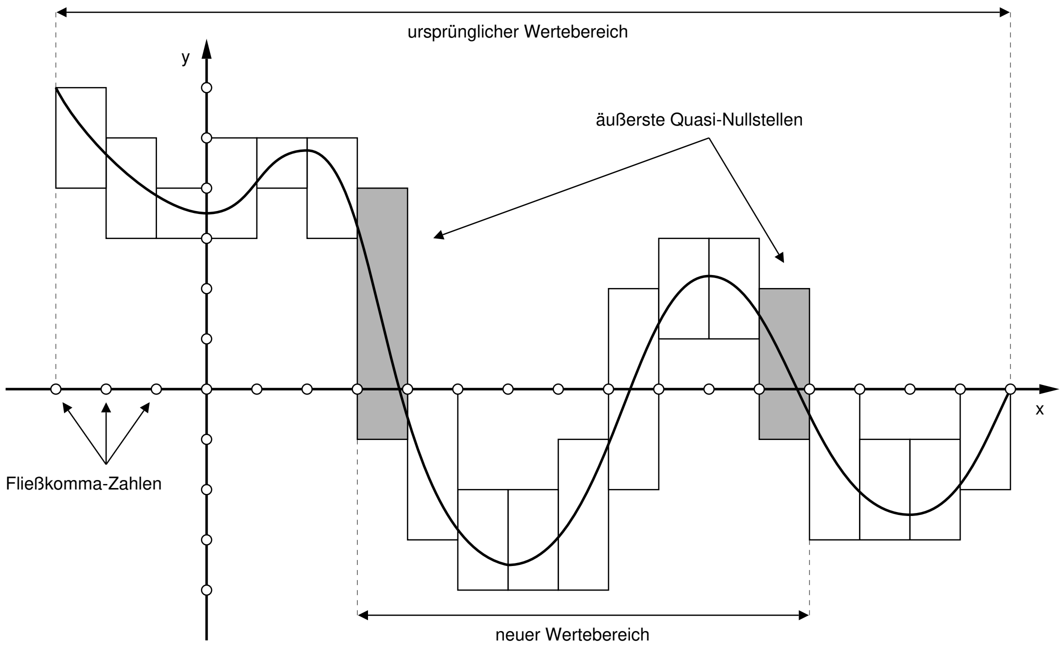


Abbildung 5.25: Bestimmung der äußersten Quasi-Nullstellen (vgl. Benhamou et al. 1999b, S. 14)

eine Annäherung an eine Nullstelle berechnet wird. Benhamou et al. (1994a) nennen diese Approximation „Quasi-Nullstelle" und bezeichnen damit, dass eine Quasi-Nullstelle aufgrund der begrenzten Fließkomma-Präzision nicht von einer Nullstelle unterschieden werden kann (vgl. Benhamou et al. 1994a, S. 131).

Um Box-Konsistenz zu erreichen, muss jeweils die äußerste linke und rechte Quasi-Nullstelle berechnet werden (siehe Abbildung 5.25). Weil das Newton-Intervallverfahren allein dazu nicht ausreicht, wird ein internes Splitting vorgenommen, so dass die Suche nach Nullstellen auf unterschiedliche Bereiche fokussiert werden kann. Jedes Intervall wird dazu in zwei gleich große Subintervalle zerteilt. Wird in dem linken Teil keine Quasi-Null gefunden, wird dieser Teil „abgeschnitten" (engl. *pruned*) und die Suche in dem verbleibenden rechten Teil fortgesetzt, wie in Abbildung 5.26 auf der nächsten Seite anhand der beispielhaft dargestellten Suchreihenfolge zu sehen ist.

5.3.6.2 Erweiterungen zur Box-Konsistenz

Wenn einzelne, punktgenaue Lösungen benötigt werden, bietet es sich an, ein übergeordnetes binäres Such- bzw. Splitting-Verfahren einzusetzen. Van Hentenryck et al. (1995, 1997) stellen den *Branch & Prune*-Algorithmus `Newton` vor, mit dem alle isolierten Lösungen eines ICSPs aufgefunden werden können. Der Algorithmus kann als globales Suchverfahren charakterisiert werden, welches Box-Konsistenz als Filtermechanismus einsetzt. Nachdem die Wertebereiche durch Box-Konsistenz so weit wie möglich eingeschränkt worden sind,

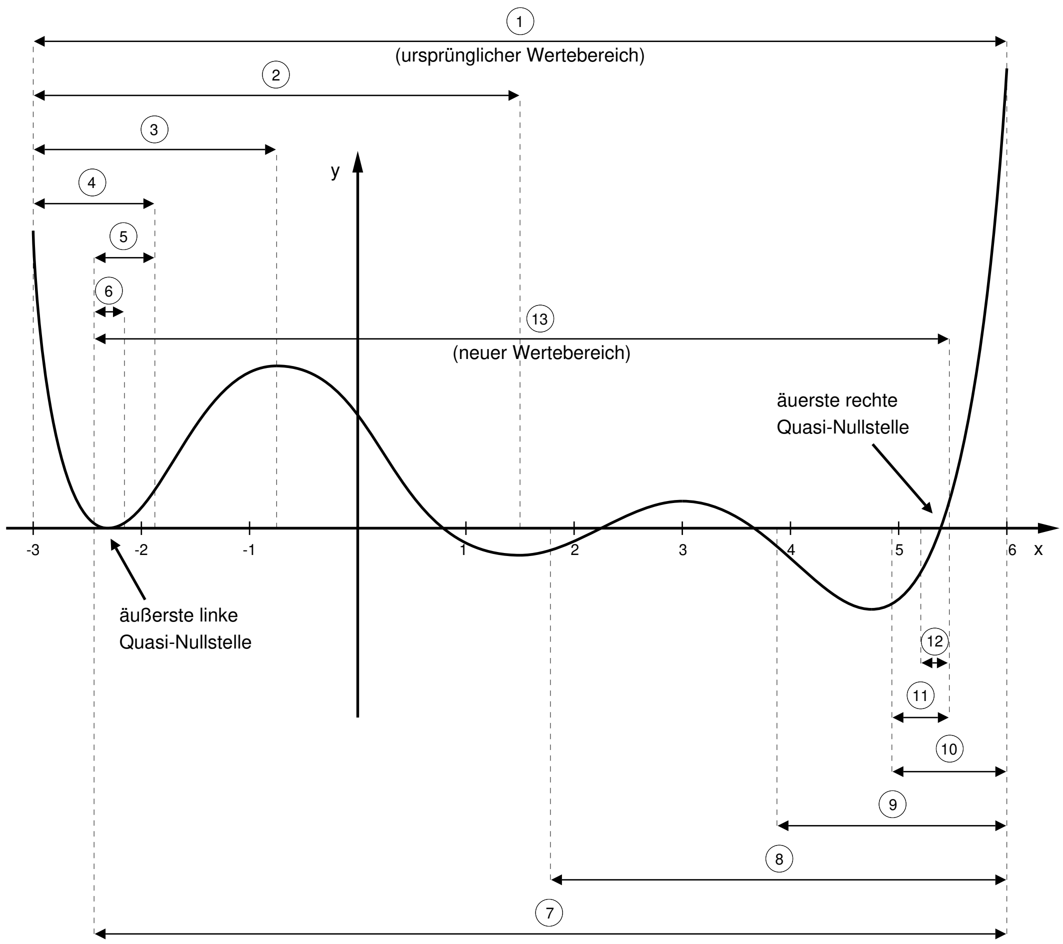


Abbildung 5.26: Berechnung von Box-Konsistenz (vgl. Benhamou 2002, S. 48)

wird ein *Branching*-Mechanismus eingesetzt, um einzelne Lösungen zu isolieren. Bereiche, die keine Lösung enthalten, werden „abgeschnitten“ bzw. verworfen.

Eine weitere Verbesserung der Box-Konsistenz findet sich bei Puget und Van Hentenryck (1996, 1998) bzw. Collavizza et al. (1998, 1999). Das beschriebene Verfahren eignet sich, mit dem entsprechenden Aufwand, zum Erreichen eines höheren Konsistenzgrades, genannt *Box(2)-Konsistenz* bzw. *Bound-Konsistenz*.[108] Hier wird das Prinzip zum Erreichen von 3B-Konsistenz auf Box-Konsistenz angewendet, d. h. es wird überprüft, ob Box-Konsistenz jeweils bzgl. der oberen und unteren Intervallgrenze für das gesamte ICSP hergestellt werden kann (vgl. Abschnitt 5.3.5.2, S. 160).

[108] Während Dechter und Rossi (2003); Marriott und Stuckey (1999) mit *bounds consistency* eine der Kantenkonsistenz angenäherte und auf die Intervallgrenzen bezogene Konsistenz benennen, die ausschließlich für finite Domänen und spezielle (*global*) Constraints eingesetzt wird, Russel und Norvig (2002); Silaghi et al. (2001); Vu et al. (2002, 2003c) unter dem Begriff *bound consistency* jegliche auf die Intervallgrenzen bezogene Konsistenzen unabhängig von der Domäne zusammenfassen, wird von Collavizza et al. (1998, 1999) mit *bound consistency* eine spezielle Version der Box-Konsistenz benannt.

Neben dem verbesserten Algorithmus `HC-4` zur Herstellung von Hull-Konsistenz ohne Zerlegung in primitive Constraints (vgl. Abschnitt 5.3.5.3, S. 161), wird darauf aufbauend von Benhamou et al. (1999a) ein effizienterer Algorithmus `BC-4` entwickelt. `BC-4` basiert auf einer Kombination von `HC-4` und `BC-3`, wie der Newton-basierte Algorithmus zur Herstellung von Box-Konsistenz von Benhamou et al. (1999a) in Anlehnung an `AC-3` genannt wird.

Unter Optimierungsaspekten ebenfalls interessant ist die Arbeit von Granvilliers et al. (1999). Das hier vorgestellte Verfahren stellt Box_φ-Konsistenz her (mittels $\mathtt{BC}_\varphi$), ein Konsistenzgrad, der aufgrund von Beschränkungen bei der Berechnung von Näherungswerten grundsätzlich schwächer ist als Box-Konsistenz, dafür allerdings effizienter berechnet werden kann. Der eingesetzte Algorithmus $\mathtt{BC}_\varphi$ ist adaptiv, d. h. er passt seine Beschränkung dynamisch an, so dass u. U. ein Konsistenzgrad erreicht werden kann, der äquivalent zur Box-Konsistenz ist.

5.3.7 2^k-Baum-Methode

Djamila Sam-Haroud und *Boi V. Faltings* stellen ein Verfahren vor, mit dem globale Konsistenz in ICSPs hergestellt werden kann (vgl. Haroud und Faltings 1994; Sam-Haroud 1995). Der Lösungsraum wird dazu in einer hierarchischen Zerlegung in Form von *2^k-Bäumen* (engl. *2^k-trees*) repräsentiert. 2^k-Bäume sind eine Repräsentationsform, die üblicherweise in der Bildverarbeitung Anwendung findet. Der Lösungsraum wird dazu in drei Kategorien eingeteilt und farblich markiert:

1. Bereiche, die ausschließlich Lösungen enthalten (*weiß*),
2. Bereiche, die Lösungen enthalten, aber auch Punkte, die nicht Teil der Lösung sind (*grau*), und
3. Bereiche, die keine Lösung enthalten (*schwarz*).

Die grauen Bereiche, die sowohl Lösungen als auch Elemente enthalten, die nicht Teil einer Lösungen sind, werden in neue Unterbereiche zerteilt und wiederum klassifiziert. Dies geschieht so lange, bis eine definierte Präzisionsgrenze erreicht wird, d. h. die zerteilten Bereiche eine bestimmte Größe erreicht haben. Der Lösungsraum wird bei diesem Vorgehen quasi durch eine Menge konsistenter Boxen „aufgefüllt“. Das Verfahren ist daher eher für die Berechnung von Lösungs*räumen* anstatt für punktgenaue Lösungen vorgesehen.[109]

5.3.7.1 Konsistenz durch 2^k-Bäume

Sam-Haroud und Faltings setzen die Anwendung von totalen Constraints voraus, d. h. sämtliche Relationen bzgl. einer Menge von Variablen werden jeweils in einem einzigen Constraint zusammengefasst (vgl. Definition 4.1.2, S. 51). In Bezug auf 2^k-Bäume gestaltet sich diese Berechnung einfach, da lediglich die einzelnen 2^k-Bäume erstellt und hiervon die

[109] Beispielsweise für Anwendungen aus dem Bereich Diagnose und Design, in denen unterbestimmte Probleme (engl. *under-constrained problems*) durch Ungleichheits-Constraints auftreten können.

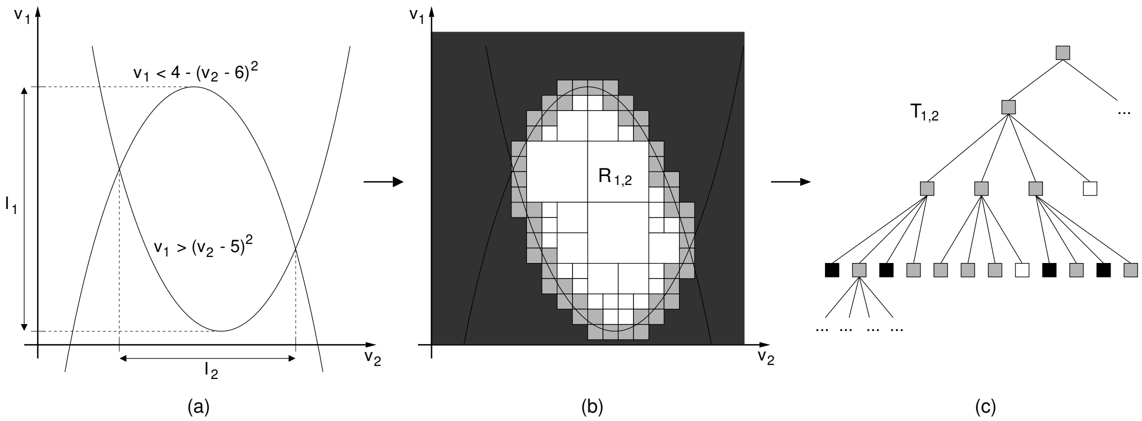


Abbildung 5.27: Beispiel für die Generierung eines 2^k-Baums (vgl. Haroud und Faltings 1994, S. 41 ff.)

Schnittmenge gebildet werden muss (vgl. Sam-Haroud und Faltings 1996a, S. 95). Der resultierende Baum beschreibt den Lösungsraum des benötigten totalen Constraints. In Abbildung 5.27 ist beispielhaft dargestellt, wie ein 2^k-Baum generiert wird:

Beispiel 5.3.10 *Die beiden Constraints aus Abbildung 5.27(a) bilden zusammen die Relation $R_{1,2}$, die den Lösungsraum für v_1 und v_2 beschreibt. Da die Constraints Ungleichungen sind, wird der Lösungsraum durch die Fläche der sich überlappenden Graphen gebildet. Projiziert auf die Achsen des Koordinatensystems, ergeben sich die gültigen Wertebereiche I_1 und I_2 für v_1 und v_2 bzgl. der vorhandenen Constraints. In 5.27(b) ist zu sehen, wie der Lösungsraum von $R_{1,2}$ durch eine hierarchische, binäre Zerlegung der Wertebereiche approximiert werden kann, und dies in 5.27(c) zu einem Quartärbaum (engl. quadtree) $T_{1,2}$ führt.*[110]

Die entstehenden 2^k-Bäume $T_{i,j}$ repräsentieren jeweils die Relationen $R_{i,j}$. Auf ihnen können Weiterentwicklungen des Waltz-Filteralgorithmus zur Herstellung von Kantenkonsistenz, Pfadkonsistenz, etc. angewendet werden. In bestimmten Fällen, für konvexe, binäre Constraints, führt Pfadkonsistenz innerhalb polynomialer Laufzeitkomplexität gleichzeitig zu globaler Konsistenz.[111] Pfadkonsistenz wird durch die in Abschnitt 5.2.3.4 auf Seite 101 f. beschriebenen Operationen *composition* und *intersection* erreicht. Sam-Haroud und Faltings (1996a, S. 98) stellen dies wie folgt dar:

$$T'_{i,j} = T_{i,j} \oplus \prod_{i,j} (T_{i,k} \otimes T_{k,k} \otimes T_{k,j})$$

Ebenso kann bei Bedarf niedrigere Konsistenz hergestellt werden. Da Pfadkonsistenz eine Generalisierung der Kantenkonsistenz ist, kann analog Kantenkonsistenz folgendermaßen erreicht werden:

[110] 2^k-Bäume: binäre Relationen führen zu Quartärbäumen, ternäre Relationen zu Oktärbäumen (engl. *octree*), usf. (vgl. Haroud und Faltings 1994, S. 42; Sam-Haroud und Faltings 1996a, S. 93)

[111] Analog strenge 5-Konsistenz für konvexe, ternäre Constraints.

$$T'_{i,i} = T_{i,i} \oplus \prod_{i} (T_{i,j} \otimes T_{j,j})$$

Während die *intersection*-Operation $\oplus$ die Überschneidung der Wertebereiche impliziert, was sich innerhalb der 2^k-Baum-Repräsentation wiederum problemlos durch Bildung der logischen Schnittmenge der betreffenden Bäume durchführen lässt, muss zur Bestimmung der „Pfade“ im Constraint-Netz durch die *composition*-Operation $\otimes$ eine komplexere Umwandlung vorgenommen werden. Anstatt die *composition*, wie bei den klassischen Pfadkonsistenz-Algorithmen, durch binäre Matrizen-Multiplikation zu berechnen, werden in Bezug auf 2^k-Bäume die k-dimensionalen Constraints durch die Projektion $\prod$ in den $(k+1)$-dimensionalen Raum und wieder zurück in die k-te Dimension projiziert. Die Regeln dieser Projektionen führen in diesem Fall zur Realisierung des *composition*-Operators. Gegeben die Ordnung $weiß < grau < schwarz$ sind die folgenden Regeln anzuwenden (vgl. Sam-Haroud und Faltings 1996a, S. 98):

i. $color(node_1 \oplus node_2) = Max(color(node_1), color(node_2))$

ii. $color(node_1 \otimes node_2) = Max(color(node_1), color(node_2))$

iii. $color(\prod(node)) = Min(color(node_i))$
wobei $node_i$ alle Knoten sind, die $\prod(node)$ als Facette besitzen

In Abbildung 5.28 auf der nächsten Seite ist ein Würfel zu sehen, dessen 3-dimensionale Knoten aus deren 2-dimensionalen Facetten abgeleitet werden können. Umgedreht ist es möglich, Informationen über eine Facette zu erhalten, indem die 3-dimensionale Ausprägung über eine ihrer Facetten projiziert wird. Die Operatoren zur Herstellung von Kantenkonsistenz, Pfadkonsistenz und weiteren Generalisierungen benötigen somit ausschließlich logische anstatt numerische Operationen zu ihrer Durchführung (vgl. Haroud und Faltings 1994, S. 43; Sam-Haroud und Faltings 1996a, S. 98).

5.3.7.2 Erweiterungen des Ansatzes

Durch die Repräsentation als 2^k-Baum können Constraints mit beliebiger Stelligkeit beschrieben werden. Die Komplexität steigt allerdings mit zunehmender Stelligkeit signifikant an, im *worst-case* exponentiell. Sam-Haroud und Faltings (1996a, S. 94 f.) weisen deshalb darauf hin, wie n-äre Constraints in ein System äquivalenter ternärer Constraints umgewandelt werden können, und behandeln in Folge ausschließlich binäre bzw. ternäre Constraints. Zur effizienteren Verarbeitung dieser Constraints wird von Sam-Haroud und Faltings (1996b, S. 413 ff.) eine neue Konsistenz eingeführt, $(3,2)$-*relational consistency*, die ausreichend ist, für ternäre Constraints globale Konsistenz zu garantieren, dabei allerdings weniger aufwendig ist, als die sonst benötigte strenge 5-Konsistenz. $(3,2)$-*relational consistency* garantiert, dass für jedes Triplett ternärer Relationen $R_1(v_1, v_4, v_5)$, $R_2(v_2, v_4, v_5)$ und $R_3(v_3, v_4, v_5)$ über zwei gemeinsame Variablen,[112] bei einer konsistenten

[112] Wobei v_4 und v_5 identisch sein können.

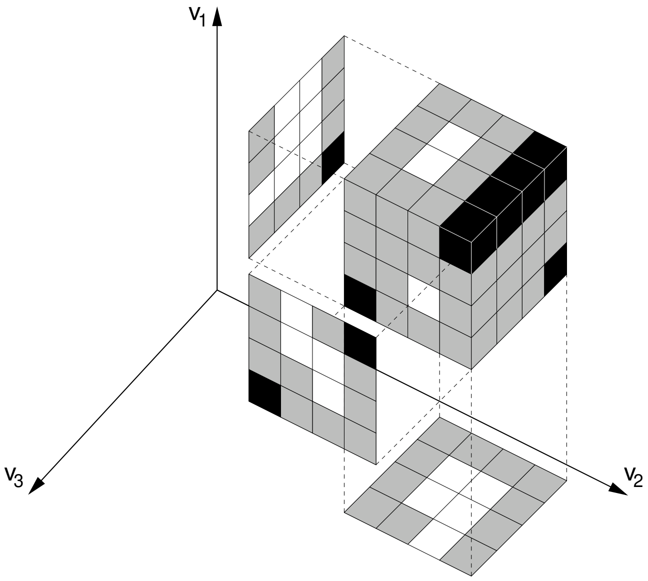


Abbildung 5.28: Projektion von Facetten (vgl. Sam-Haroud und Faltings 1996a, S. 97)

Belegung von v_1, v_2, v_3 die Variablen v_4, v_5 so belegt werden können, dass die Relationen R_1, R_2 und R_3 erfüllt werden. Für jedes konvexe, ternäre Constraint-Netz ist $(3, 2)$-*relational consistency* äquivalent mit globaler Konsistenz. Sam-Haroud und Faltings verallgemeinern dieses Prinzip zur $(r, r\text{-}1)$-*relational consistency* für r-stellige Constraints bzw. Constraint-Gruppen.

Außerdem ist es möglich, die Konvexitätsbedingung auf eine partielle Konvexität abzuschwächen, so dass für eine größere Anzahl Problemstellungen bereits durch Pfadkonsistenz bzw. $(3, 2)$-*relational consistency* globale Konsistenz erreicht werden kann. Weil die Voraussetzung von konvexen, *non-disjunctive* Constraints sehr streng ist, definieren Sam-Haroud und Faltings *axis-convexity* für binäre Constraints, eine schwächere Form von Konvexität, die ausreichend ist, globale Konsistenz sicherzustellen (vgl. Sam-Haroud und Faltings 1996a, S. 101 ff.; Sam-Haroud und Faltings 1996b, S. 417 ff.). *Axis-convexity* bzgl. einer Variablen v_k bzw. (v_k)-*convexity* bezieht sich auf die Konvexität der *Projektionen* von Verbindungen zwischen Punkten aus einer Relation R, die sich parallel zur Achse von v_k befinden. Da *axis-convexity* ausschließlich die Konvexität der parallelen Verbindungen voraussetzt, ist diese Form weniger restriktiv als die Forderung nach allgemeiner Konvexität, d. h. von Projektionen beliebiger Verbindungen zwischen Punkten aus R. Auch das Prinzip der *axis-convexity* für binäre Constraints wird von Sam-Haroud und Faltings zur sog. $(v_1, \ldots, v_k)$-*convexity* für n-äre Constraints verallgemeinert.

Der Ansatz, die Berechnung von Lösungsräumen anstatt punktgenaue Lösungen konsequenter zu unterstützen, wird in diversen Veröffentlichungen weiter verfolgt (vgl. Silaghi et al. 2001; Vu et al. 2003c). In Anlehnung an die 2^k-Baum-Methode wird dabei die Einteilung des Lösungsraums in drei Kategorien beibehalten. Zum Einsatz kommen außerdem erweiterte Splitting-Strategien, die zwischen Gleichheits- und Ungleichheits-Constraints unterscheiden und unterschiedliche Heuristiken einsetzen, sowie Kombinationen mit Verfahren zum Herstellen von Hull-Konsistenz zur konventionellen Einschränkung von Wertebereichen.[113]

5.4 Zusammenfassung und Diskussion

Für klassische Constraint Satisfaction Probleme mit finiten Domänen stehen eine Vielzahl an Constraint-Lösungsmethoden und Heuristiken zur Verfügung. Grundsätzlich wird zwischen Konsistenz- und (systematischen) Suchverfahren unterschieden. Durch die ausschließliche Anwendung von Konsistenzverfahren ist es außer in Ausnahmefällen nicht möglich, Lösungen für CSPs zu bestimmen. Stattdessen werden Konsistenzverfahren häufig als Preprozessing bzw. während eines Suchverfahrens, durch dessen Ausführung die eigentlichen Lösungen generiert werden, als *look-ahead*-Mechanismus eingesetzt. Um eine weitere Optimierung des Laufzeitverhaltens zu erreichen, können Suchverfahren durch Heuristiken zur Variablen- und Werteauswahl unterstützt werden.

Eine besondere Stellung nehmen binäre CSPs ein, da viele der existierenden Lösungsalgorithmen auf binäre Constraints beschränkt sind. Verbunden wird dies häufig mit dem Hinweis, dass sich die Verfahren grundsätzlich auf n-äre CSPs verallgemeinern lassen. Aufgrund der hohen Komplexität n-ärer Constraints sind tatsächlich nur wenige Algorithmen verallgemeinert worden. Die Möglichkeit zur Binärisierung n-ärer Constraint-Netze verbessert die Situation nicht wirklich, da durch die Umwandlung in eine extensionale Repräsentation in Form von „umfassenden Variablen" ein sehr hoher Platzbedarf verbunden mit relativ hohem Berechnungsaufwand entsteht. Die Binärisierung n-ärer CSPs, um anschließend binäre Lösungsalgorithmen verwenden zu können, ist daher nur für überschaubare Problemstellungen zu empfehlen.

Im Rahmen dieser Arbeit soll als Grundausstattung für ein Constraint-Framework eine Basis an soliden Lösungsverfahren entstehen, die ein stabiles Laufzeitverhalten für möglichst viele Problemstellungen bieten. Verfahren zur Berechnung globaler Konsistenz sind i. A. zu aufwendig und zu ineffizient. Kantenkonsistenz dagegen ist ein weit verbreitetes Verfahren, welches einen angemessenen Kompromiss zwischen Berechnungsaufwand und der Menge der gefilterten Werte darstellt. Aufgrund der notwendigen extensionalen Repräsentation für den `AC-4`-Algorithmus und höher (durch *support*-Einträge), sowie aufgrund des durchschnittlich besseren Laufzeitverhaltens (vgl. Wallace 1993) sollte dem `AC-3`-Algorithmus bzw. einer der weiterentwickelten Varianten (`AC-8`, `AC-2000`, `AC-2001`, `AC-3.1`, `AC-3`$_d$) der Vorzug gegeben werden. Pfadkonsistenz ist für viele Problemstellungen zu aufwendig. Für die vorhandenen Algorithmen ist zudem eine Matrix-Repräsentation der Constraints erforderlich, was das Verfahren aufgrund des entstehenden Ressourcenbedarfs ungeeignet für umfangreiche Problemstellungen macht. Durch die Matrix-Repräsentation sind Algorithmen zur Herstellung von Pfadkonsistenz überdies auf binäre Constraints beschränkt. Ausgeschlossen werden soll ein derartiger Constraint-Solver allerdings nicht, denn u. U. ist ein Konsistenzgrad größer als Kantenkonsistenz für bestimmte, überschau-

[113] Für die Algorithmen von Silaghi et al. (2001) und Vu et al. (2003c) wird zum Herstellen von 3B-Konsistenz der ILOG Solver eingesetzt.

bare Problemstellungen durchaus sinnvoll in Form einer Effizienzverbesserung.[114] Weitere Konsistenzarten sind Verallgemeinerungen (inverse Konsistenz, (i, j)-Konsistenz) oder spezialisierte Varianten, die bspw. dazu dienen, möglichst frühzeitig Inkonsistenzen festzustellen (LAC) oder geringfügig höhere bzw. niedrigere Konsistenzgrade anbieten als die Standardverfahren (DAC, DPC, RPC, SAC, SRPC, adaptive Konsistenz). Diese Konsistenzen können bei speziellen Problemstellungen sinnvoll sein, sie werden allerdings nicht schwerpunktmäßig in dieser Arbeit behandelt.

Neben Filter- bzw. Konsistenzalgorithmen müssen durch die Constraint-Komponente verschiedene Suchverfahren angeboten werden, durch die Lösungen ermittelt werden können, sofern diese existieren. Für einfache Problemstellungen sollte, um unnötigen Overhead zu vermeiden, einfaches, chronologisches Backtracking nutzbar sein. Außerdem sollte für die Verarbeitung komplexer Constraint-Netze eine (konfliktbasierte) Backjumping-Variante angeboten werden, denn Backjumping ist, im Gegensatz zu anderen Suchverfahren mit *look-back*-Strategie wie Backchecking und Backmarking, mit dynamischen und damit leistungsfähigen Heuristiken zur Variablenordnung einsetzbar. Diese Suchverfahren sollten außerdem mit Filteralgorithmen als *look-ahead*-Mechanismus kombiniert angeboten werden (FC, MAC). Zur Variablenordnung sollte eine Kombination des *fail-first*-Prinzips und der *maximum-degree-ordering*-Heuristik ein stabiles Laufzeitverhalten bieten. Heuristiken zur Werteauswahl bringen, begründet durch den hohen Aufwand jeden einzelnen Wert zu betrachten, im Schnitt weniger Gewinn und müssen daher nicht vorrangig behandelt werden. Die Gefahr, durch derartige Ordnungsheuristiken unnötigen Overhead zu produzieren, ist ungleich größer. Werteordnungsheuristiken erhöhen die Effizienz von Suchverfahren zudem ausschließlich dann, wenn nicht alle Lösungen eines Problems benötigt werden.

Grundsätzlich sollten die oben genannten Verfahren auch für n-äre Constraint-Netze anwendbar sein, auch wenn der Einsatz aufgrund der hohen Komplexität derartig formulierter Problemstellungen nicht zu empfehlen ist.

Die Domänen reellwertiger algebraischer Constraints innerhalb von Intervall Constraint Satisfaction Problemen werden in Form von Intervallen mit oberer und unterer Grenze definiert. Kombinatorische Verfahren zur Aufzählung möglicher Lösungen versagen hier. Allerdings lassen sich, basierend auf intervallarithmetischen Grundlagen, Konsistenzalgorithmen wie aus Abschnitt 5.2 auf Seite 83 ff. bekannt in abgewandelter Form effizient anwenden. Die Wertebereiche der Constraint-Variablen werden hierbei möglichst weit eingeschränkt, ohne mögliche Lösungen des Problems zu verlieren. Durch ein Splitting der Wertebereiche können in einem weiterführenden Suchvorgang punktgenaue Lösungen ermittelt werden. Aufgrund einer Vielzahl von Splitting-Möglichkeiten führt dieses Verfahren bei umfangreicheren Problemen allerdings schnell zu einer kombinatorischen Explosion.

Basierend auf dem Waltz-Filteralgorithmus zur Herstellung von Kantenkonsistenz wurde von Davis (1987) das sog. Label Inference zur Einschränkung von Intervallgrenzen entwickelt. Weiterentwicklungen davon, die eine Zerlegung von Constraints in primitive

[114] In Abschnitt 5.3.7.1 auf Seite 167 ff. werden darüber hinaus die Grundlagen zur Herstellung von Pfadkonsistenz für die Verarbeitung von reellwertigen Intervalldomänen eingesetzt.

Constraints vornehmen und dadurch eine generelle Anwendbarkeit ermöglichen, sind die Toleranzpropagation von Hyvönen (1992) und Hull-Konsistenz bzw. 2B-Konsistenz von Lhomme (1993).[115] Während im Rahmen der Toleranzpropagation von Hyvönen (1992) Möglichkeiten aufgezeigt werden, mit denen durch (*dynamic*) Splitting und der Eliminierung von Zyklen im Constraint-Netz ggf. globale Konsistenz erreicht werden kann, werden von Lhomme (1993) im Rahmen der Hull-Konsistenz höhere Konsistenzgrade durch 3B- bzw. kB-Konsistenz definiert.

Eine weiterführende Methode von Benhamou et al. (1994a) zur Herstellung von Box-Konsistenz setzt numerisch-mathematische Verfahren ein. Eingebettet in einen Waltz-ähnlichen Filteralgorithmus wird das Newton-Intervallverfahren, zur Einschränkung der Intervallgrenzen genutzt. Auch hier lassen sich ggf. durch Splitting kanonische Lösungen generieren.

Ein weiterer Ansatz, die 2^k-Baum-Methode von Haroud und Faltings (1994), ist in der Lage, für ununterbrochene Intervalle bzw. für Intervalle, die speziellen Konvexitätsbedingungen genügen, effizient globale Konsistenz sicherzustellen. Im Gegensatz zu Waltz-basierten Verfahren garantiert die 2^k-Baum-Methode durch ein binäres Suchverfahren Konvergenz. Sie hat allerdings im Gegensatz zu anderen numerischen Verfahren nicht die Berechnung eines einzigen, optimalen Fixpunktes zum Ziel, sondern ist eher dazu geeignet Lösungsräume zu berechnen, die von aus Ungleichungen bestehenden Constraint-Problemen gebildet werden. Für diese Fälle lässt sich eine kompakte Repräsentation des Lösungsraums berechnen.[116]

Für die zu entwickelnde Komponente zur Verarbeitung von Constraints mit infiniten Domänen in EngCon ist es vorrangig erforderlich, Algorithmen zum Einschränken der Intervallgrenzen zur Verfügung zu stellen. Berechnungsverfahren explizit für kanonische Lösungen stehen weniger im Fokus. Die Wertebereichseinschränkungen dagegen sollten so weit wie möglich vorgenommen, d. h. die Lösungen so dicht wie möglich umschlossen, werden. Dies lässt sich für eine Vielzahl an Problemstellungen effizient mit Verfahren ähnlich dem Waltz-Filteralgorithmus erreichen. Die (lokale) Toleranzpropagation und die Algorithmen zur Herstellung von Hull-Konsistenz (2B-Konsistenz, 3B-Konsistenz) implementieren dies für Intervalldomänen. Während für die Toleranzpropagation bzw. für Hull-Konsistenz lediglich eine Zerlegung der Constraints vorgenommen werden muss, ist für die Herstellung von Box-Konsistenz die Implementierung komplexer mathematischer Operationen und ein automatischer Gleichungsumformer erforderlich, durch den die Constraints in ein durch das Newton-Intervallverfahren verarbeitbares Format gebracht werden müssen. Für die 2^k-Baum-Methode ist eine komplexe Datenrepräsentation und eine aufwendige Projektion zu implementieren. Im Gegensatz zu diesen Verfahren, die in ihrer Umsetzung sehr aufwendig sind, ist eine Implementierung der Toleranzpropagation bzw. von Algorithmen zur Herstellung von Hull-Konsistenz innerhalb eines vertretbaren Zeitaufwands möglich.

[115] Der durch lokale Toleranzpropagation erreichte Konsistenzgrad entspricht 2B-Konsistenz.

[116] Für punktgenaue Lösungen, d. h. für Constraint-Gleichungen anstatt Ungleichungen, müssten, bezogen auf Beispiel 5.3.10 auf Seite 168, die exakten Schnittpunkte der Graphen, anstatt der umschlossenen Fläche, mittels Zerlegung in einen 2^k-Baum approximiert werden. Dies ist durch die der 2^k-Baum-Methode eigenen Repräsentation des Lösungsraums intuitiv weniger effizient durchführbar, als mit anderen Verfahren.

Aus Gründen der Komplexität sollten die in dieser Arbeit umgesetzten Verfahren ausschließlich konvexe Intervalldomänen verarbeiten. Zum einen steigt bei der Berechnung mit diskontinuierlichen Intervallen, d. h. mit Vereinigungsmengen einzelner Teilintervalle, der Berechnungsaufwand stark an, und zum anderen existieren für Berechnungen mit ununterbrochenen Intervallen bereits Werkzeuge, insbesondere für die Programmiersprache Java (z. B. IAMath, vgl. Abschnitt 4.5.2, S. 72).

Im folgenden Kapitel wird ein Konzept vorgestellt, mit dem sich die unterschiedlichen, hier dargelegten Constraint-Lösungsverfahren für finite und infinite Domänen innerhalb eines hybriden Frameworks flexibel austauschen und kombinieren lassen.

Teil III

Konzeption und Realisierung

Kapitel 6

Ein hybrides Framework zur heterogenen Constraint-Verarbeitung

Simple things should be simple, complex things should be possible.

Alan Kay

Im folgenden Kapitel wird das Konzept für ein hybrides Constraint-System entwickelt. Eingebettet in eine modulare und wiederverwendbare Framework-Architektur ist es strategiebasiert, wodurch sich Lösungsverfahren flexibel einsetzen und austauschen lassen. Zur Verarbeitung von hybriden Constraint-Problemen wird die Funktion eines Meta-Constraint-Solvers zum Lösen von heterogenen Constraints spezifiziert sowie mehrere Szenarien zur Umsetzung des Systems aufgezeigt.

6.1 Einführung

In vielen Anwendungsbereichen lassen sich Problemstellungen als Constraint-Problem formulieren. Aufgrund einer Vielzahl unterschiedlicher Domänen und der Vielfältigkeit der Anwendungsszenarien ist es notwendig, an die jeweilige Domäne angepasste Constraint-Lösungsverfahren einzusetzen. Zur Behandlung unterschiedlicher Constraint-Domänen innerhalb eines strukturbasierten Konfigurierungswerkzeugs ist eine Kooperation mehrerer Constraint-Solver sowohl für finite als auch infinite Domänen erforderlich. Diese Kooperation muss durch eine Komponente geleistet werden, mit der sich unterschiedliche Constraint-Solver je nach Bedarf und domänenspezifisch einsetzen lassen. Die Modularität des Entwurfs ist dabei entscheidend für die Austauschbarkeit einzelner Komponenten.

Das vorherige Kapitel hat gezeigt, dass aufgrund der Vielzahl möglicher Problemspezifikationen und der großen Zahl existierender Lösungsverfahren mit ihren unterschiedlichen

Eigenschaften es außerdem sinnvoll ist, innerhalb einer Domäne je nach Problemstellung entsprechende Lösungsverfahren einzusetzen: Um eine möglichst hohe Effizienz bei der Problemlösung zu erreichen, sollten die Lösungsverfahren problemabhängig anwendbar sein. Auch hier stellt sich die Anforderung an die Modularität: Um flexibel die jeweils geeigneten Verfahren einsetzen zu können, werden insbesondere gekapselte Schnittstellen und ein Mechanismus benötigt, die einen problemlosen Wechsel einzelner Komponenten bzw. Constraint-Solver erlauben. Dadurch würde zudem die Erweiterbarkeit des Systems sichergestellt werden, indem zukünftig benötigte, spezielle Lösungsverfahren an eine einheitliche Schnittstelle angebunden werden können.

Die zu erstellende und in EngCon zu integrierende Constraint-Komponente sollte im Detail die folgenden Anforderungen erfüllen:

- Die zu entwickelnde Komponente sollte einen modularen Aufbau und einheitliche Schnittstellen für unterschiedliche Lösungsverfahren aufweisen.

- Constraint-Lösungsverfahren müssen sich flexibel einbinden bzw. problemabhängig austauschen lassen.

- Das System muss *hybrid* sein, d. h. neben finiten Domänen müssen infinite Domänen in Form von reellwertigen Intervallen unterstützt werden.

- Der inkrementelle Aufbau des Constraint-Netzes innerhalb einer interaktiv durchgeführten, strukturbasierten Konfigurierung muss unterstützt werden.

Nachfolgend wird ein Konzept vorgestellt, mit dem sich die genannten Anforderungen, die sich hinsichtlich einer Integration in das Konfigurierungswerkzeug EngCon stellen, erfüllen lassen. In diesem Kapitel wird dazu eine Constraint-Komponente spezifiziert, welche im Folgenden mit YACS benannt wird. Der Name YACS steht für *Yet Another Constraint Solver*. YACS ist allerdings mehr als nur ein einzelner Constraint-Solver.[1] Es bezeichnet vielmehr ein hybrides System zum flexiblen Einsatz von Constraint-Lösungsverfahren für finite und infinite Domänen. Sie sind eingebettet innerhalb einer modularen Framework-Architektur.

6.2 Der Framework-Ansatz

Durch das Aufkommen von objektorientierten Sprachen und objektorientierter Programmierung (OOP) entstanden Ansätze, welche verstärkt die Steigerung der Wiederverwendbarkeit von einmal entwickelten Software-Komponenten zum Ziel hatten (vgl. Johnson 1997a, b). Im Besonderen sind dies objektorientierte Software-Frameworks, in denen ein Rahmenwerk für die Bewältigung eines bestimmten Aufgabenspektrums bereitgestellt wird. Sie bieten eine Architekturhilfe beim Aufteilen des Entwurfs in abstrakte Klassen

[1] Der Namensgebung von YACS liegt damit eine gewisse Ironie zugrunde, denn YACS ist gerade nicht ein „gewöhnlicher“ Constraint-Solver. Vielmehr ermöglicht das YACS-Framework die flexible Kooperation und Kombination von Lösungsalgorithmen zu Constraint-Solvern mit neuen bzw. erweiterten Eigenschaften.

und deren Spezialisierungen sowie bei der Definition ihrer Zuständigkeiten und Interaktionen. Ein objektorientiertes Framework wird für eine bestimmte Anwendung spezialisiert, indem der Entwickler anwendungsspezifische Unterklassen für abstrakte Framework-Klassen erstellt (vgl. Gamma et al. 1996, S. 37).

Objektorientierte Constraint-Frameworks im Speziellen dienen dazu, OOP-Sprachen und Techniken zur Constraint-Verarbeitung zu verbinden und in unterschiedlichen Szenarien nutzbar zu machen. Wie in Abschnitt 4.5.3 auf Seite 74 bereits angesprochen, ist ein Constraint-Framework eine Möglichkeit, Constraints als Inferenz-Mechanismus unabhängig von einer konkreten Domäne, z. B. der (logischen) Constraint-Programmierung, nutzbar für unterschiedliche Anwendungen zu machen. Ein Framework bietet hierfür allgemeine Mechanismen, die zur Nutzung durch eine bestimmte Anwendung an die jeweils spezielle Problemstellung angepasst werden können (vgl. Roy et al. 2000, S. 1 f.).

Durch ein Constraint-Framework wird ein allgemeiner Kontrollzyklus vorgegeben, in den unterschiedliche Lösungsverfahren je nach Bedarf eingebunden werden können. Allgemeine Verfahren zur Constraint-Verarbeitung sind innerhalb eines Frameworks in einer (erweiterbaren) Bibliothek bereits enthalten. Neben einer wiederverwendbaren Architektur stellt ein Framework somit ebenfalls wiederverwendbaren Code zur Verfügung. Die Architektur eines Frameworks sollte dabei eine einfache Nutzung garantieren, und in diesem Fall die komplexen Mechanismen des CSP-Formalismus vor dem Benutzer weitestgehend verbergen (vgl. Roy et al. 2000, S. 4).

Wie im vorherigen Kapitel 5 dargestellt, existieren sehr unterschiedliche Konsistenz- und Such- bzw. Splitting-Verfahren, Heuristiken und Kombinationen dieser. Es gibt kein optimales Constraint-Lösungsverfahren für alle Problemstellungen, sondern stattdessen nur unterschiedlich gut geeignete Verfahren für unterschiedliche Probleme und Domänen. Um je nach Problemstellung flexibel geeignete Lösungsverfahren einsetzen zu können, bietet sich für die zu entwickelnde Constraint-Komponente YACS aus mehreren Gründen ein objektorientierter Framework-Ansatz an:

- Da für die strukturbasierte Konfigurierung mit EngCon ein hybrides Constraint-System, sowohl für finite als auch infinite Constraint-Domänen benötigt wird, sind unterschiedliche Constraint-Solver erforderlich.

- Neben implementierten Constraint-Solvern können in ein derartiges Framework über eine Schnittstellenspezifikation bestehende Constraint-Systeme eingebunden werden, die, wenn sie die Anforderungen von EngCon auch nicht vollständig erfüllen, für bestimmte Problemstellungen hinreichend (oder gar notwendig) sind.

- Ein Framework bietet die notwendige (wiederverwendbare) Umgebung zur einfachen Implementierung neuer Lösungsverfahren bzw. zur einfachen Integration von Fremdsystemen.

- Es bietet darüber hinaus den Vorteil der modularen Erweiterbarkeit und allgemeine Schnittstellen für den flexiblen Einsatz in unterschiedlichen Anwendungen auch außerhalb der strukturbasierten Konfigurierung.

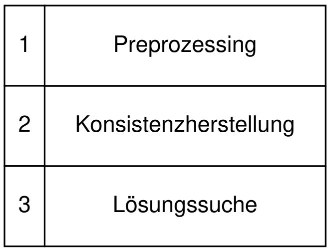


Abbildung 6.1: Aufbau einer Constraint-Lösungsstrategie

Der Framework-Ansatz ist ein vielversprechendes Konzept im Bereich der OOP, welches weite Verbreitung gefunden hat (vgl. Fayad und Schmidt 1997, S. 34 ff.). Neben der geforderten Modularität bietet das Framework-Konzept für die Constraint-Komponente YACS außerdem die leichte Wiederverwendbarkeit und Erweiterbarkeit der einmal erstellten Software-Architektur.

6.3 Constraint-Lösungsstrategien

Flexibilität bzgl. der einzusetzenden Lösungsverfahren kann durch ein Konzept von modularen und austauschbaren Constraint-Lösungsstrategien erreicht werden. Zur Strukturierung des Constraint-Lösungsvorgangs wird dieser Prozess in drei Phasen eingeteilt:

1. Preprozessing
2. Konsistenzherstellung
3. Lösungssuche

Diese Phasen spiegeln sich innerhalb von Constraint-Lösungsstrategien wieder (vgl. Abbildung 6.1). In der ersten Phase wird ein Preprozessing des Constraint-Problems vorgenommen. Dies kann sich z. B. auf die Binärisierung eines Constraint-Netzes oder die Zerlegung von Constraints in primitive Constraints beziehen, um anschließend darauf aufbauende Lösungsalgorithmen anwenden zu können. In der zweiten Phase werden Filter- bzw. Konsistenzalgorithmen zur Einschränkung der Domänen der Constraint-Variablen angewendet. Da dies allein i. A. nicht zu einer Lösung des Constraint-Problems führt, können in einer dritten Phase Suchverfahren zum Auffinden von Lösungen in den reduzierten Wertebereichen eingesetzt werden.[2]

Zu beachten ist, dass in jeder Phase mehrere Einträge innerhalb einer Lösungsstrategie existieren können. So ist es z. B. möglich, mehrere Preprozessing-Schritte auf ein Problem anzuwenden, bevor Verfahren aus der nächsten Phase zum Einsatz kommen. Dies gilt ebenso für Konsistenz- und Suchverfahren.

Während es für Konsistenzverfahren durchaus sinnvoll erscheint bspw. Knotenkonsistenz herzustellen, bevor ein Algorithmus zum Herstellen von Kantenkonsistenz eingesetzt

[2]In Bezug auf Problemstellungen mit intervallwertigen Domänen dienen Splitting-Verfahren der Suche nach kanonischen Lösungen (vgl. Abschnitt 5.3.2, S. 150).

1	-
2	Knotenkonsistenz
3	Forward Checking

Strategie 1

1	Binärisierung
2	(1) Knotenkonsistenz (2) Kantenkonsistenz
3	konfliktbasiertes Backjumping

Strategie 2

1	Zerlegung in primitive Constraints
2	Hull-Konsistenz
3	-

Strategie 3

Abbildung 6.2: Beispiele für Constraint-Lösungsstrategien

wird, erschließt sich der Sinn mehrerer Einträge im Fall von Suchverfahren nicht sofort. Für den Fall allerdings, dass für eine geeignete Anwendung aus Effizienzgründen anstatt systematischer Suchverfahren lokale bzw. stochastische Suchverfahren zum Einsatz kommen, kann hierdurch die Unvollständigkeit lokaler Verfahren abgemildert werden: Wenn ein (lokales) Suchverfahren keine Lösung für ein Constraint-Problem gefunden hat, sich aber mehrere Einträge in der Phase für die Lösungssuche befinden, kann entsprechend das nächste Suchverfahren angewendet werden.[3]

Ebenso ist es möglich, dass für eine Phase innerhalb einer Strategie keine Einträge existieren. Nicht für alle Konsistenz- und Suchverfahren ist ein Preprozessing erforderlich. Gleichfalls kann ein Suchverfahren auch ohne vorherigen Filteralgorithmus angewendet werden, insbesondere wenn das Suchverfahren bereits Filtermechanismen enthält. Sind für eine Anwendung keine exakten Lösungen sondern nur eingeschränkte Wertebereiche erforderlich, kann auf ein Suchverfahren in der dritten Phase verzichtet werden. Mehrere Beispiele für mögliche Constraint-Lösungsstrategien sind in Abbildung 6.2 zu sehen.

Die Verwaltung derartiger Constraint-Lösungsstrategien muss von einer Komponente vorgenommen werden, die in der Lage ist, aufgrund einer klaren Spezifikation die Zuordnung der jeweiligen Strategien zu einzelnen Teilproblemen des gesamten Constraint-Problems vornehmen zu können. Wie dies im Detail geschieht, wird im nächsten Abschnitt verdeutlicht.

6.4 Ein hybrides Constraint-System

Ein hybrides Constraint-System zeichnet sich dadurch aus, dass es in der Lage ist, ein hybrides Constraint Satisfaction Problem zu verarbeiten:

Definition 6.4.1 (Hybrides Constraint Satisfaction Problem)
Ein System zur Verarbeitung eines hybriden Constraint Satisfaction Problems H wird durch die Angabe von sieben Komponenten

$$H = (C, S, \delta, V_{fd}, D_{fd}, V_{int}, D_{int})$$

[3] Lokale Suche wird aufgrund der Unvollständigkeit dieser Verfahren nicht im Zusammenhang der strukturbasierten Konfigurierung mit EngCon umgesetzt.

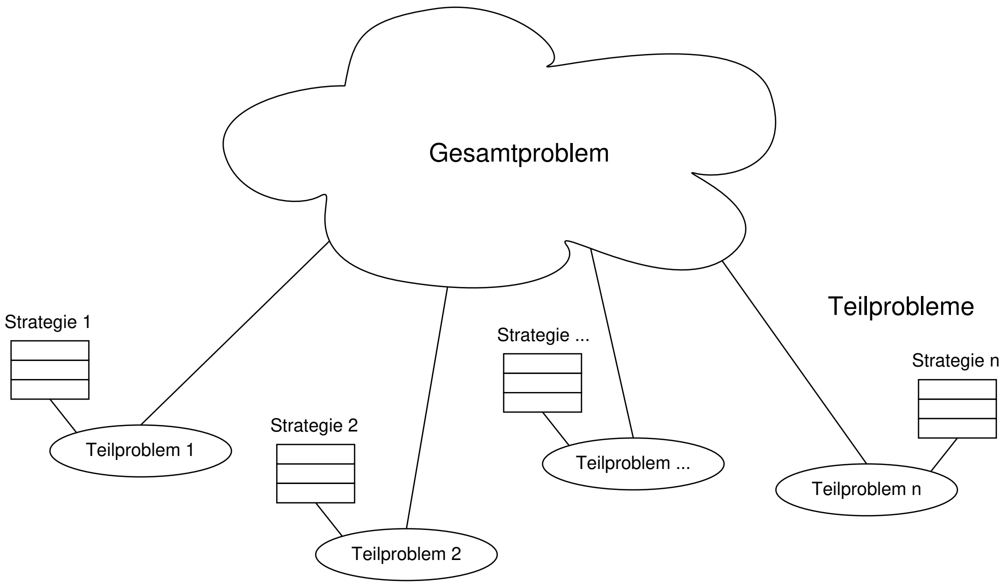


Abbildung 6.3: Zuständigkeiten unterschiedlicher Strategien für Teilbereiche des Constraint-Problems

beschrieben. Dabei ist $C = \{C_1, \ldots, C_m\}$ *eine endliche Menge von Constraints und* $S = \{S_1, \ldots, S_n\}$ *eine endliche Menge von Constraint-Lösungsstrategien. Die Funktion* δ *ordnet jedem Constraint* C_i, $i \in \{1, \ldots, m\}$, *eine eindeutige Strategie* S_j, $j \in \{1, \ldots, n\}$, *zu:*

$$\delta : C_i \rightarrow S_j.$$

Der Bezeichner V_{fd} *steht für eine endliche Menge von FD-Variablen* $\{v_1, \ldots, v_k\}$, *mit denen die Wertebereiche* $D_{fd} = \{D_1, \ldots, D_k\}$ *mit* $\{v_1 : D_1, \ldots, v_k : D_k\}$ *assoziiert sind. Ebenso sind die Intervallvariablen* $V_{int} = \{v_1, \ldots, v_l\}$ *mit den Wertebereichen* $D_{int} = \{D_1, \ldots, D_l\}$ *mit* $\{v_1 : D_1, \ldots, v_l : D_l\}$ *assoziiert. Jedes Constraint* C_i *setzt eine Teilmenge der Variablen aus* V_{fd} *und* V_{int} *zueinander in Relation und beschränkt deren gültige Wertekombinationen auf eine Teilmenge des kartesischen Produkts ihrer Wertebereiche.*

Ein hybrides CSP vereinigt somit Constraints über Variablen mit finiten und infiniten Domänen. Jedem Constraint ist eine Lösungsstrategie zu dessen Verarbeitung zugeordnet. Dies führt zu einer Aufteilung des ursprünglichen Constraint-Problems in unterschiedliche Teilprobleme, welche durch die jeweils zuständige Constraint-Lösungsstrategie definiert werden (vgl. Abbildung 6.3).[4] Zur „Überlappung“ unterschiedlicher Teilprobleme kann es kommen, wenn eine Variable in strukturell unterschiedlichen Teilproblemen auftaucht, d. h. in Constraints, die unterschiedlichen Strategien zugeordnet sind.

[4]Teilprobleme entstehen, indem mehrere Constraints derselben Strategie zugeordnet werden.

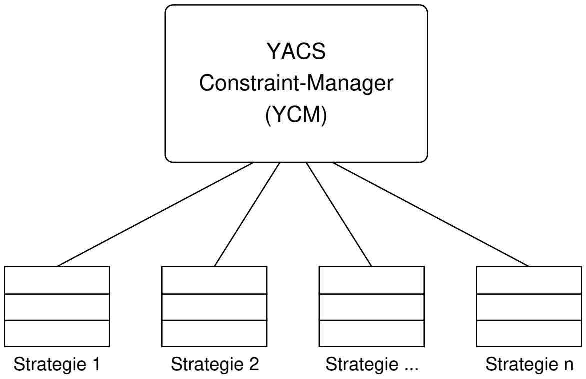


Abbildung 6.4: Verwaltung von Lösungsstrategien durch den Constraint-Manager

Wie der Ablauf zur Verarbeitung von einem derartigen, hybriden Constraint-Problem aussieht und wie mögliche Überlappungen der Teilprobleme unterschiedlicher Constraint-Lösungsstrategien behandelt werden können, wird in den folgenden Abschnitten erläutert.

6.4.1 Ausführungsmodelle

Die derzeitige Aufgabe des internen Constraint-Managers innerhalb von EngCon beschränkt sich auf die Verwaltung des Constraint-Netzes in der Form, dass der Reihe nach immer wieder die in EngCon zum Einsatz kommenden Constraint-Klassen (Tupel-Constraints, Java-Constraints und Funktions- bzw. Prädikat-Constraints, vgl. Abschnitt 3.6.2, S. 36), mit Hilfe der zur Verfügung stehenden Constraint-Lösungskomponenten propagiert werden, bis keinerlei Wertebereichsänderungen bei den beteiligen Constraint-Variablen mehr auftreten. In einer sequentiellen Abfolge werden jeweils die Instanzen der drei Constraint-Klassen als „Block“ unabhängig von den übrigen propagiert. Es werden temporär drei lokale, unabhängige Constraint-Netze generiert, die wiederum von den jeweiligen Constraint-Lösungskomponenten verarbeitet werden. In einer „Metapropagation“ werden die Wertebereichsänderungen der Teilpropagationen in das übergeordnete Constraint-Netz von EngCon übertragen. Wertebereichsänderungen innerhalb eines Sub-Netzes werden so in den jeweils anderen Sub-Netzen berücksichtigt.

Benötigt wird eine Komponente an der Schnittstelle zwischen der vorhandenen, internen Constraint-Verwaltung von EngCon und dem YACS-Framework, der die Verwaltung der neu hinzukommenden unterschiedlichen Constraint-Netze, der Constraint-Lösungsstrategien, der Constraint-Solver und letztendlich der Steuerung des Lösungsprozesses obliegt. Diese Verwaltungs- und Steuerungsaufgabe wird von dem „YACS Constraint-Manager“ (YCM) wahrgenommen (vgl. Abbildung 6.4).

Von dem aufrufenden System, in diesem Fall EngCon, erhält der Constraint-Manager YCM die Informationen, welche Constraints mit welcher Strategie aufzulösen sind. Der Constraint-Manager aktiviert die entsprechenden Constraint-Lösungskomponenten und übergibt in der jeweiligen Bearbeitungsphase das entsprechende Constraint-Netz zur Verarbeitung an die dafür vorgesehene Komponente.

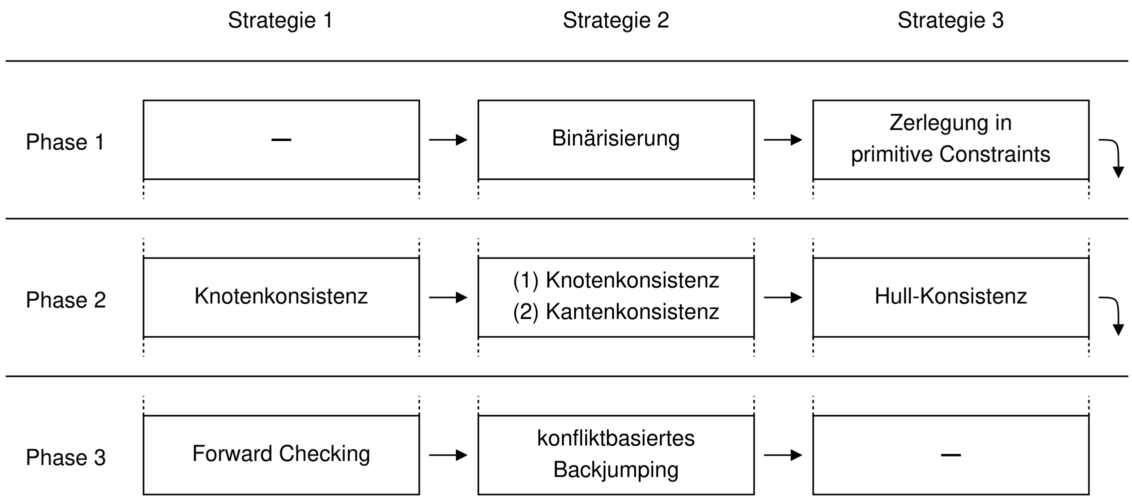


Abbildung 6.5: Beispiel für phasenweisen Lösungsprozess

Der Prozess des Constraint-Lösens ist analog zum Aufbau der Constraint-Lösungsstrategien in drei Phasen unterteilt. In jeder Phase werden sequentiell jeweils die Constraint-Netze aller Strategien der Reihe nach bearbeitet.[5] Das heißt die in den Strategien angegebenen Constraint-Verfahren werden in den entsprechenden Phasen auf die zugehörigen Constraint-Netze angewendet (vgl. Abbildung 6.5):

- **Phase 1 (Preprozessing):** Das jeweilige Constraint-Netz wird wenn möglich vollständig konvertiert oder es wird festgestellt, dass es sich mit den gegebenen Algorithmen nicht umformen lässt.
- **Phase 2 (Konsistenzherstellung):** Es wird solange propagiert, bis keine Änderungen der Wertebereiche mehr eintreten (d. h. Konsistenz hergestellt ist) oder eine Inkonsistenz auftritt.
- **Phase 3 (Lösungssuche):** Die Suchalgorithmen finden die geforderten Lösungen oder stellen fest, dass keine Lösung existiert.

Eine Ausnahme, wenn sich z. B. ein Constraint-Netz nicht konvertieren lässt oder eine Inkonsistenz festgestellt wird, führt zum Abbruch des Lösungsvorgangs. Ein Beispiel für einen Lösungsprozess mit unterschiedlichen Strategien, deren Constraint-Verfahren in den jeweiligen Phasen der Reihe nach abgearbeitet werden, ist in Abbildung 6.5 zu sehen. Für eine Realisierung dieses Strategiekonzepts kommen zwei unterschiedliche Szenarien in Frage:

Szenario 1: Ein vereinfachtes Szenario sieht vor, dass für jede Wissensbasis jeweils lediglich eine Strategie für finite und eine für intervallwertige Domänen vorgegeben

[5] An dieser Stelle bietet sich Optimierungspotential bzgl. einer verteilten Constraint-Verarbeitung unterschiedlicher Teilprobleme.

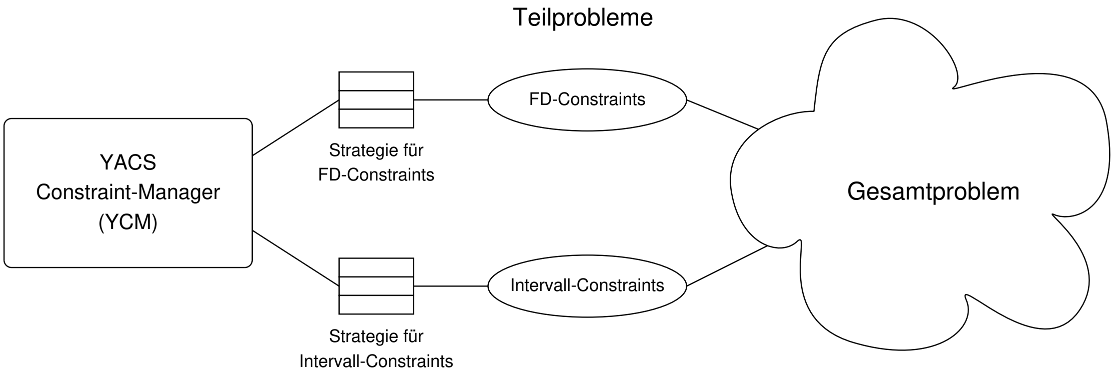


Abbildung 6.6: Vereinfachtes Szenario mit zwei Lösungsstrategien

werden kann (vgl. Abbildung 6.6). Um die Unabhängigkeit von YACS sicherzustellen, sollte dies außerhalb und separat von der EngCon-Wissensbasis geschehen. In diesem Fall würde der Constraint-Manager von YACS die Constraint-Lösungsstrategien auslesen und anschließend anwenden. Der Constraint-Lösungsprozess gestaltet sich in diesem Szenario folgendermaßen:

- Zuerst werden für die beiden Constraint-Netze (finite und infinite Domänen) die in den Strategien definierten Preprozessingschritte ausgeführt.
- Als nächstes werden für beide Constraint-Netze die Konsistenzverfahren so lange angewendet, bis keine Wertebereichsänderungen mehr auftreten.
- Als letztes werden die Lösungsverfahren der Strategien auf beide Constraint-Netze angewendet.

Abschließend erfolgt die Rückmeldung der neuen Wertebereiche der Constraint-Variablen sowie die möglichen Lösungen an das Constraint-System von EngCon.

Vorteile dieses einfachen Szenarios sind die schnelle und einfache Realisierbarkeit und der geringe Overhead, der in Bezug auf die Verwaltung innerhalb des Constraint-Managers von YACS entsteht. Flexibilität bzgl. des Einsatzes von Constraint-Lösungsverfahren ist dadurch gewährleistet, dass für jede Wissensbasis und dem darin enthaltenen Konfigurierungs- und Constraint-Problem erneut entschieden werden kann, welche Strategien respektive Constraint-Verfahren angewendet werden sollen. Nachteilig hinsichtlich der Flexibilität wirkt sich dieses Szenario möglicherweise bei komplexen Constraint-Problemen aus. Innerhalb von umfangreichen Problemstellungen kann es sinnvoll sein, Unterprobleme, d. h. Teile des Constraint-Netzes, aus Effizienzgründen mit speziellen Constraint-Lösungstechniken zu verarbeiten. Für andere Bereiche des Constraint-Problems wiederum können dieselben Constraint-Verfahren unnötigen Overhead bedeuten.

Ob dieser Weg eine angemessene Umsetzung darstellt, ist abhängig von der Komplexität des Constraint-Netzes innerhalb der Wissensbasis. Bei überschaubaren Pro-

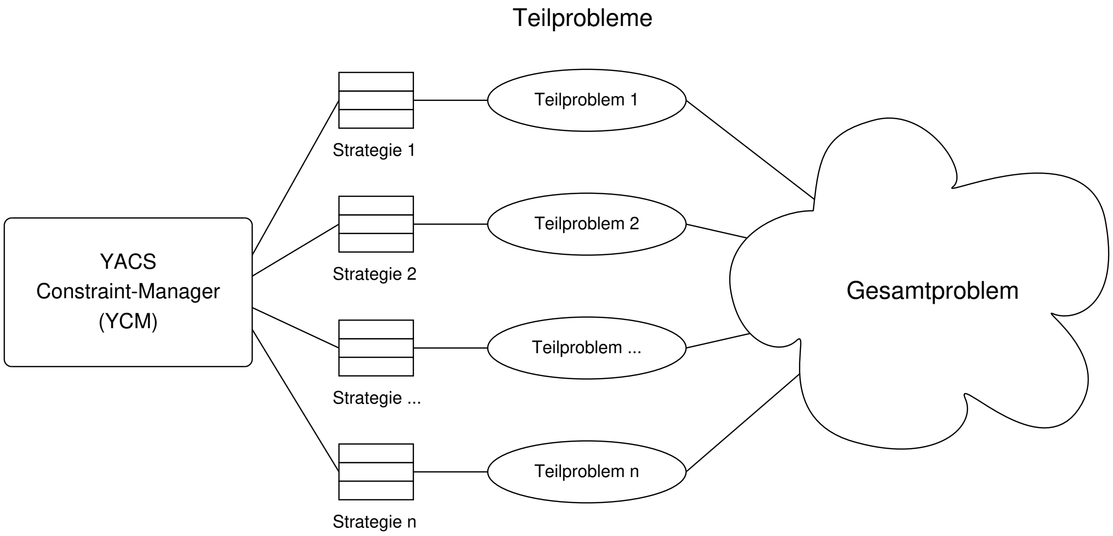


Abbildung 6.7: Erweitertes Szenario mit beliebig vielen Lösungsstrategien

blemstellungen werden ggf. nicht mehr als zwei Strategien (je eine für finite und infinite Domänen) benötigt.

Szenario 2: Erweiterte Flexibilität wird erreicht, wenn in der Wissensbasis für jedes Constraint der Name einer entsprechenden Strategie zu dessen Lösung angegeben werden kann. Der Name der jeweiligen Strategie entspricht dabei einem eigenen Teilbereich des ursprünglichen Constraint-Problems. Jede Strategie ist für die Verarbeitung eines (Sub-)Constraint-Netzes vorgesehen. Bei der Initialisierung wird durch den Constraint-Manager von YACS sichergestellt, dass alle geforderten Strategien existieren und angewendet werden können.

Die Strategien werden separat von EngCon definiert. EngCon übergibt die Constraints jeweils mit dem Namen der zugehörigen Strategie an den YACS Constraint-Manager. Dieser generiert daraus die unterschiedlichen Constraint-Netze und wendet die entsprechenden Constraint-Lösungsstrategien an (vgl. Abbildung 6.7):

- Zuerst werden für sämtliche Constraint-Netze die in den Strategien definierten Preprozessingschritte ausgeführt.
- Als nächstes werden für alle Constraint-Netze die jeweils entsprechenden Konsistenzverfahren so lange angewendet, bis in den jeweiligen Netzen keine Wertebereichsänderungen mehr auftreten.
- Als letztes werden die Lösungsverfahren der Strategien auf die jeweils entsprechenden Constraint-Netze angewendet.

Abschließend erfolgt auch hier die Rückmeldung der neuen Wertebereiche der Constraint-Variablen sowie die möglichen Lösungen an das Constraint-System von EngCon.

Da für jedes Constraint theoretisch eine eigene Constraint-Lösungsstrategie angegeben werden kann, ist in diesem Szenario die maximale Flexibilität bzgl. des Einsatzes unterschiedlicher Constraint-Lösungsverfahren für unterschiedliche Topologien von Constraint-Netzen sichergestellt. Nachteilig ist der höhere Aufwand für die Realisierung dieses Szenarios. Zudem ist mit höherem Overhead als in dem erstgenannten Szenario zu rechnen. Dies betrifft sowohl die Verwaltung im Constraint-Manager von YACS, als auch die Erstellung und Pflege der Wissensbasis von EngCon.

Der zusätzliche Overhead in diesem Szenario ist allerdings nicht sehr hoch und kann weiter verringert werden, indem bei überschaubaren Problemstellungen weniger Strategien eingesetzt werden. So gesehen ist dieses Szenario in Abhängigkeit von der Problemstellung „skalierbar", und kann ggf. auf das Szenario 1 mit lediglich zwei unterschiedlichen Strategien reduziert werden. Das Szenario 2 stellt demnach eine Verallgemeinerung von Szenario 1 dar, und umgedreht das Szenario 1 eine Spezialisierung von Szenario 2.

Unabhängig von den diesen beiden Szenarien ist optional aus Gründen der Vereinfachung die Anwendung von *Default Strategien* für finite und infinite Domänen in Erwägung zu ziehen. Default Strategien könnten immer dann angewendet werden, wenn durch den Nutzer keine spezielle Constraint-Lösungsstrategie spezifiziert wurde.

Werden innerhalb eines strategiebasierten Constraint-Systems unterschiedliche Teilprobleme von unterschiedlichen Constraint-Solvern verarbeitet, so stellt sich die Frage, wie zu verfahren ist, wenn Constraints über Variablen definiert sind, die in unterschiedlichen Teilproblemen auftauchen. Innerhalb eines hybriden Systems ist darüberhinaus relevant, ob die Überlappung von Teilproblemen mit unterschiedlichen Wertedomänen (finit/infinit) zulässig ist oder nicht. Im Folgenden werden diese Fragen untersucht und Lösungsansätze aufgezeigt.

6.4.2 Hybridizität versus Heterogenität

Üblicherweise wird ein Constraint-Problem von einem einzigen Constraint-Solver mit einem eindeutigen Namensraum und eindeutigen Variablenbelegungen bzw. Wertebereichen bearbeitet. Unterschiedliche Constraint-Lösungsstrategien allerdings bedingen getrennt voneinander zu verarbeitende Teilbereiche des ursprünglichen Constraint-Problems. Unterschiedliche Constraint-Solver kooperieren in diesem Fall, um gemeinsam mögliche Lösungen für das Constraint-Problem zu generieren.

Ein wesentliches Problem in diesem Szenario ist, wie aus den einzelnen Teillösungen vollständige Gesamtlösungen (globale Lösungen) generiert werden können. Ein in diesem Zusammenhang zentraler Punkt betrifft die Frage, ob die Constraint-Netze der unterschiedlichen Constraint-Solver innerhalb von YACS denselben Namensraum aufweisen oder nicht, d. h. die Frage ob „Überlappungen" der Constraint-Netze möglich sein sollen oder nicht, und wie diese zu behandeln sind. Überlappungen treten an den Stellen auf, an denen dieselben Variablen in den Constraint-Netzen mehrerer Strategien auftauchen.

Das Szenario mit voneinander getrennten, disjunkten Constraint-Netzen mit jeweils eigenem Namensraum ohne Überlappungen wird *lokale Sicht* genannt:

Definition 6.4.2 (lokale Sicht)
Für eine Menge von Problemen $\{P_1, \ldots, P_m\}$ *und einer Menge von Variablen* $\{v_1, \ldots, v_n\}$ *bedeutet eine lokale Sicht, dass die Mengen der in zwei unterschiedlichen Problemen* P_1 *und* P_m *enthaltenen Variablen* $\{v_{1_1}, v_{1_2}, \ldots\}$ *und* $\{v_{m_1}, v_{m_2}, \ldots\}$ *disjunkt sind.*

Mit einer lokalen Sicht wäre das System hybrid, ließe aber keine Überlappungen zwischen den einzelnen Teilproblemen und den unterstützten Domänen zu. Eine derartige Verarbeitung innerhalb von YACS würde eine Vereinfachung bedeuten. Gleichzeitig wird das Problem allerdings verlagert, wenn für die übergeordnete Anwendung globale Lösungen für das Constraint-Problem benötigt werden. Hierfür ist ein einheitlicher Namensraum erforderlich. Existiert nur ein einziger Namensraum und sind Überlappungen von Constraint-Netzen unterschiedlicher Strategien zulässig, so wird dies *globale Sicht* genannt:

Definition 6.4.3 (globale Sicht)
Für eine Menge von Problemen $\{P_1, \ldots, P_m\}$ *und einer Menge von Variablen* $\{v_1, \ldots, v_n\}$ *bedeutet eine globale Sicht, dass die Mengen der in zwei unterschiedlichen Problemen* P_1 *und* P_m *enthaltenen Variablen* $\{v_{1_1}, v_{1_2}, \ldots\}$ *und* $\{v_{m_1}, v_{m_2}, \ldots\}$ *dieselben Elemente beinhalten können.*

Ein System mit globaler Sichtweise ist nicht nur hybrid, es müsste zudem heterogene Constraints und damit heterogenes Constraint-Lösen unterstützen.

Für eine genauere Untersuchung erfolgt an dieser Stelle wiederum eine phasenweise Betrachtung des Constraint-Lösungsprozesses. Dies geschieht jeweils in Bezug auf voneinander getrennte Namensräume (lokale Sicht) und in Bezug auf denselben Namensraum für alle beteiligten Constraint-Verfahren (globale Sicht):

- **Phase 1 (Preprozessing)**

  Die Preprozessing-Phase ist weitgehend unkritisch bzgl. überlappender Constraint-Netze von kooperierenden Constraint-Solvern in unterschiedlichen Strategien:

  Lokale Sicht: Jedes Constraint-Netz wird für sich und unabhängig vom restlichen Problem konvertiert und damit für den weiteren Lösungsprozess vorbereitet.

  Globale Sicht: Umformungsprozesse fügen häufig Variablen und zusätzliche Constraints zum ursprünglichen Problem hinzu. Beispielsweise werden sowohl bei einer Binärisierung (umfassende) Variablen hinzugefügt (finite Domänen), als auch bei einer Zerlegung der Constraints in primitive Constraints (infinite Domänen). Diese Konvertierungen des Constraint-Netzes sind unkritisch, da bei Überschneidungen der Constraint-Netze der Zugriff auf die ursprünglichen Variablen gesichert bleibt. Wenn Umformungsprozesse allerdings Variablen ersetzen oder der Zugriff auf die ursprünglichen Variablen nicht möglich sein sollte, wird dadurch in Kauf genommen, dass Einschränkungen von deren Wertebereichen YACS-intern nicht übergreifend registriert und propagiert werden können.

- **Phase 2 (Konsistenzherstellung)**

Die Phase der Konsistenzherstellung profitiert von einem einheitlichen Namensraum der Constraint-Netze aller Strategien. Es wird direkt innerhalb von YACS eine höhere Filterung inkonsistenter Werte erreicht, wodurch für nachfolgende Suchverfahren weniger Suchaufwand notwendig wird:

Lokale Sicht: Wertebereichseinschränkungen werden von jeder Strategie für das dem entsprechenden Teilproblem zugehörige Constraint-Netz separat vorgenommen. Es werden daher i. A. weniger inkonsistente Werte entfernt bzw. die Intervallgrenzen weniger stark beschränkt, als wenn Einschränkungen im vollständigen Constraint-Problem propagiert werden. Sollte in den Strategien kein abschließendes Suchverfahren definiert sein, können beim Zurückschreiben der eingeschränkten Wertebereiche in das interne Constraint-System von EngCon unterschiedliche Filterstärken anhand unterschiedlich stark eingeschränkter Wertebereiche registriert werden. An dieser Stelle kann durch EngCon eine erneute Propagation initiiert werden, wobei für jede Variable der jeweils kleinste Wertebereich berücksichtigt werden sollte, um möglichst viele inkonsistente Werte auszuschließen. In einem weiteren Durchlauf der Propagation werden diese Einschränkungen auch für andere Namensräume sichtbar.

Globale Sicht: Wertebereichseinschränkungen, die durch Konsistenzverfahren einer Strategie hervorgerufen werden, haben in diesem Fall direkt Auswirkungen auf alle überlappenden Constraint-Netze anderer Strategien. Einschränkungen werden augenblicklich in allen Constraint-Netzen sichtbar und haben entsprechende Auswirkungen auf die Filterung inkonsistenter Werte bzw. die Einschränkung der Intervallgrenzen. Sollte kein abschließendes Suchverfahren erfolgen, sind die an EngCon zurückgegebenen Wertebereiche bereits so weit wie möglich eingeschränkt. Eine erneute Propagation würde keine zusätzlichen Einschränkungen bewirken und ist damit obsolet.

- **Phase 3 (Lösungssuche)**

Werden ausschließlich Constraint-Verfahren zur Einschränkung der Wertebereiche und Herstellung bestimmter Konsistenzgrade eingesetzt, ist die Kooperation unterschiedlicher Constraint-Solver unproblematisch. Werden dagegen von der Anwendung konkrete Lösungen für das Constraint-Problem benötigt, so müssen aus den einzelnen Teillösungen in einem *Meta-Constraint-Problem* globale Lösungen generiert werden:

Lokale Sicht: Für jedes Constraint-Netz unterschiedlicher Lösungsstrategien werden wie bei der klassischen Verarbeitung von Constraint-Problemen separate Lösungen berechnet und an EngCon zurückgemeldet. Lösungsverfahren einer Strategie profitieren in diesem Fall nicht von möglichen Wertebereichseinschränkungen anderer Strategien. Es besteht zudem innerhalb von YACS keine Möglichkeit, globale Lösungen zu generieren, wenn die Constraint-Netze unterschiedlicher Strategien Überlappungen aufweisen. Der Wissensingenieur muss daher bei der Spezifikation der Wissensbasis beachten, dass ausschließlich abgeschlossene Teile des Constraint-Problems jeweils einer Strategie zugeordnet werden. Die Ergebnisse der Lösungssuche sind entsprechend lediglich lokal konsistente Teillösungen. Wurde eine Trennung der Teilprobleme beachtet, so können diese für globale Lösungen beliebig miteinander kombiniert werden. Anderenfalls wird in der Anwendung ein übergeordneter Mechanismus notwendig, der aus den separaten Teillösungen globale Lösungen generiert.

Globale Sicht: Die Wertebereiche sind durch mögliche Konsistenzverfahren in allen Constraint-Netzen gleich stark eingeschränkt worden. Im Rahmen der Lösungssuche berechnen auch hier zunächst, wie bei der lokalen Sichtweise, unterschiedliche Constraint-Solver mit unterschiedlichen Strategien für jedes Constraint-Netz separat die jeweils möglichen Teillösungen. Im Rahmen der globalen Sicht allerdings können anschließend *innerhalb* von YACS globale Lösungen für das Problem generiert werden. Dafür ist nach der Berechnung der Teillösungen eine weitere Constraint-Lösungskomponente erforderlich, welche in dem entstandenen Meta-Constraint-Problem nach möglichen Lösungen sucht. Die Menge der Lösungen für ein Teilproblem stellt analog zu einer n-dimensionalen Matrix jeweils eine vollständige Relation zwischen den beteiligten Variablen dar.[6] Jede einzelne Teillösung steht für eine gültige Belegung der Relation. Das aufzulösende Constraint-Problem ist entsprechend ein extensional repräsentiertes, n-stelliges CSP.

Während Preprozessingverfahren von der globalen Sicht i. A. nicht profitieren, ergibt sich für Konsistenzverfahren der Vorteil, dass bei überlappenden Constraint-Netzen Wertebereichseinschränkungen innerhalb von YACS global propagiert werden können, anstatt nur innerhalb des durch die jeweilige Strategie definierten Teilproblems. Ebenso ermöglicht die globale Sicht das Generieren globaler Lösungen für überlappende Constraint-Netze.

Wenn innerhalb von YACS hingegen eine lokale Sichtweise angenommen wird, führt dies zu einer Vereinfachung, da die Constraint-Verfahren jeder Strategie ausschließlich ihr zugewiesenes Teilproblem bearbeiten müssen. Übergreifende Wertebereichseinschränkungen könnten bei einer erneuten Propagierung verarbeitet werden. Globale Lösungen allerdings können von YACS in diesem Fall nur dann berechnet werden, wenn sichergestellt wird, dass sich die einzelnen Teilprobleme auch global nicht überlappen.

Werden in einer globalen Sichtweise überlappende Constraint-Lösungsstrategien, die Suchverfahren einsetzen, gleichzeitig mit solchen Strategien verwendet, in denen auf Suchverfahren verzichtet wird (Strategien, die lediglich die Wertebereiche durch Konsistenzverfahren einschränken), so muss die Lösungssuche für die betroffenen Variablen auch von einem übergeordnetem Lösungsmechanismus nicht geleistet werden. An den Stellen allerdings, wo es zu Überlappungen kommt, müssen für die betreffenden Constraints Lösungen berechnet werden, die konsistent mit den entsprechenden Wertebereichen sind.[7]

[6] Sämtliche Teillösungen – auch unterschiedlicher Strategien – welche sich auf dieselbe Variablenmenge beziehen, müssen hierbei ggf. zusammen betrachtet werden, um eine vollständige Relation mit allen Abhängigkeiten zwischen diesen Variablen zu erhalten (vgl. Definition 4.1.2, S. 51, *totales Constraint*).

[7] Durch eine erneute Propagation kann es bei Übernahme der Werte aus den Lösungsverfahren als neue Domänenwerte auch durch Konsistenzverfahren zu eindeutigen Lösungen kommen.

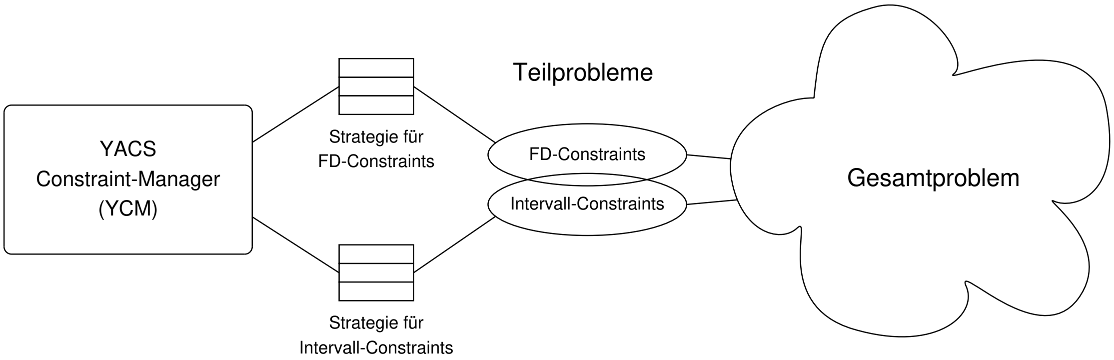


Abbildung 6.8: Einfaches Szenario mit zwei überlappenden Constraint-Netzen

Überlappungen der Constraint-Netze unterschiedlicher Strategien können das Vermischen unterschiedlicher Wertedomänen bedeuten. Während dies in Szenario 1 aufgrund der begrenzten Anzahl Strategien bei Überlappungen grundsätzlich der Fall ist (vgl. Abbildung 6.8), kann dies auch in der Verallgemeinerung des Problems in Szenario 2 gegeben sein: Bei Überlappungen ist es möglich, dass intervallwertige Domänen infiniter Variablen von diskreten Werten finiter Variablen beschränkt werden und umgekehrt (vgl. Abbildung 6.9 auf der gegenüberliegenden Seite). Das in diesem Fall vorliegende CSP ist, wie in Abschnitt 4.4.4 auf Seite 64 beschrieben, ein Mixed CSP (vgl. Gelle 1998; Gelle und Faltings 2003) bzw. ein heterogenes Constraint-Problem (vgl. Benhamou 1996):

Definition 6.4.4 (Heterogenes Constraint-Problem)
Ein heterogenes Constraint-Problem ist ein CSP, welches heterogene Constraints enthält. Heterogene Constraints sind Constraints über Variablen verschiedener Constraint-Domänen. Ein heterogenes Constraint-Problem setzt daher min. ein hybrides CSP voraus.

Die Einschränkung der Wertebereiche und die Lösungssuche hat in Bezug auf die globale Sicht analog zu anderen Überlappungen zu erfolgen. Zu beachten ist allerdings, dass in diesem Fall durch YACS automatische Konvertierungen der Wertebereiche vorgenommen werden müssen. Dies bedeutet eine Diskretisierung von intervallwertigen Domänen und umgekehrt die Propagation der Grenzen von Wertebereichen, wenn FD-Variablen von Intervall-Constraint-Solvern verarbeitet werden.[8] Wenn Konvertierungen der Wertebereiche vorgenommen werden, ist von dem Wissensingenieur zu beachten, dass nach Möglichkeit keine „ungünstigen“ Domänen konvertiert werden. Das heißt, es sollte vermieden werden, dass z. B. eine Domäne mit den Werten $\{0, 10000, 1000000\}$ in eine (konvexe) Intervalldomäne konvertiert wird. Umgedreht sollte eine Variable mit einer intervallwertigen Domäne von bspw. $[0, 1000000]$ nicht innerhalb eines FD-Constraints verwendet werden.[9]

Die Überlappung von Constraints mit finiten und infiniten Wertedomänen kann dazu führen, dass wenn eine Intervallvariable von einem FD-Constraint beschränkt wird, dies

[8]Eine intelligente Diskretisierungsmethode würde bspw. eine FD-Variable mit dem Wertebereich $\{3, 17, 27\}$, welcher durch Intervallpropagationsverfahren auf das Intervall $[3, 20]$ beschränkt wurde, bezogen

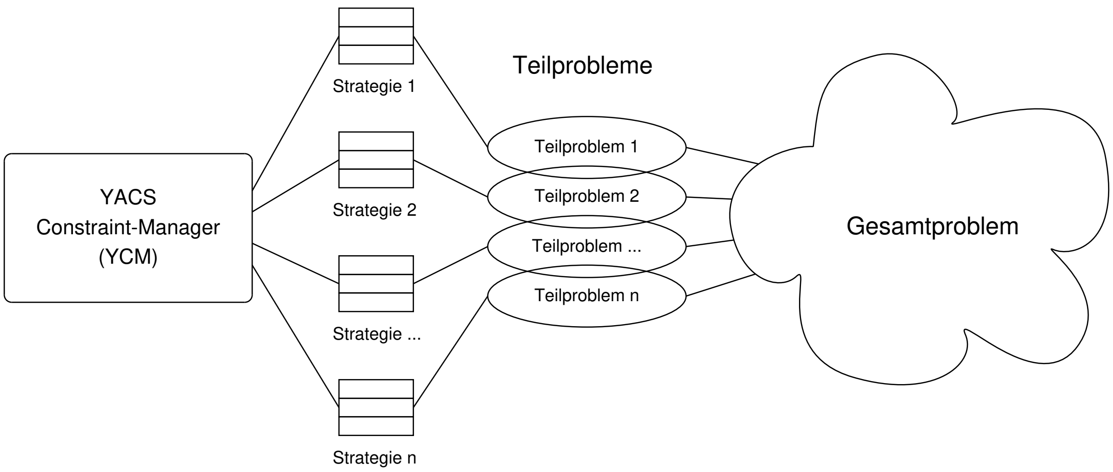


Abbildung 6.9: Erweitertes Szenario mit überlappenden Constraint-Netzen

(in Abhängigkeit von den Beschränkungen durch die Constraints) bei Auftauchen von Punktintervallen eine Kettenreaktion auslöst, an deren Ende ggf. alle Intervallvariablen auf einzelne Punktintervalle beschränkt sind. Wenn das System diskontinuierliche Intervalle unterstützt, tritt dieser Effekt bei überlappenden Constraint-Netzen verstärkt auf. Wenn hingegen ausschließlich konvexe Intervalle verarbeitet werden, tritt ein Informationsverlust auf, denn der FD-Solver hat ggf. mehr Werte aus einer Domäne herausgefiltert, als von den konvexen Intervallgrenzen umschlossen werden.

6.4.3 Heterogenes Constraint-Lösen

Für die Berechnung von globalen Lösungen ist ein übergeordneter Constraint-Lösungsmechanismus erforderlich. Wenn aus den möglicherweise teils überlappenden Teillösungen unterschiedlicher Strategien Gesamtlösungen generiert werden sollen, stellt dies ein neues kombinatorisches Problem dar: ein extensionales, n-äres *Meta CSP*:

Definition 6.4.5 (Meta Constraint Satisfaction Problem)
Ein Meta CSP besteht aus unterschiedlichen Teilproblemen $P_{Meta} = \{P_1, \ldots, P_m\}$ *und setzt eine endliche Menge von Variablen* $V = \{v_1, \ldots, v_n\}$ *mit assoziierten Wertebereichen* $D = \{D_1, \ldots, D_n\}$ *mit* $\{v_1 : D_1, \ldots, v_n : D_n\}$ *in Relation zueinander. Für jedes Teilproblem* $P_1, \ldots, P_m$ *existiert eine Menge von konsistenten Teillösungen* $T = \{T_1, \ldots, T_m\}$*, wobei jede Teillösung in* T_i*,* $i \in \{1, \ldots, m\}$*, eine Teilmenge der Variablen* $v_j \in V$*,* $j \in \{1, \ldots, n\}$ *mit einem jeweils zugeordnetem Wert* $d_j \in D_j$ *beinhaltet.*

---

auf den ursprünglichen Wertebereich auf die Werte $\{3, 17\}$ beschränken, und das Constraint anschließend für eine erneute Propagation vorsehen.

[9]Alternativ könnte in letzterem Fall zur Effizienzverbesserung innerhalb der betreffenden Constraint-Lösungsstrategie für FD-Constraints ein Intervallverfahren als Preprozessing integriert werden.

Jede Teillösung umfasst sämtliche involvierte Variablen des Teilproblems und stellt dadurch eine extensional repräsentierte Relation zwischen diesen Variablen dar.[10] Aufgabe eines übergeordneten Constraint-Lösungsverfahrens ist es, aus den n-ären Relationen konsistente Lösungen zu generieren.

Wenn durch die Constraint-Verfahren einer Lösungsstrategie im Rahmen eines Teilproblems ein Wert einer Variable aus deren Domäne entfernt wird, geschieht dies deshalb, weil dieser Wert inkonsistent mit einem Constraint ist, und auf keinen Fall Teil einer Lösung sein kann. Das heißt die Teillösungen unterschiedlicher Strategien ergänzen sich nicht, sondern schließen sich gegenseitig aus. Eine mögliche Lösung des Meta CSPs besteht deshalb darin, die Teillösungen einer Strategie jeweils mit den Teillösungen der anderen Strategien zu kombinieren und auf widersprüchliche oder konsistente Wertebelegungen zu überprüfen. Dies entspräche allerdings einem Generate-&-Test-Vorgehen (vgl. Abschnitt 5.2.4.1, S. 113).

Aus Effizienzgründen sollte stattdessen ein systematisches Suchverfahren zur Anwendung kommen. Eine vollständige Suche muss durch sukzessives, systematisches Belegen aller Variablen, durch Testen, ob die jeweils aktuelle (Teil-)Belegung von allen Teillösungen unterstützt wird und durch Backtracking im Fall von Inkonsistenzen systematisch Lösungen anhand der existierenden Relationen in den Teillösungen generieren. Das Lösungsverfahren ist umso aufwendiger, je mehr Teillösungen das Constraint-Problem aufweist. In Problemen mit niedriger Lösungsdichte, bezogen sowohl auf die Teillösungen als auch auf die Gesamtlösungen, ist die Lösungssuche effizienter. Gegebenenfalls bietet sich eine Unterstützung des Meta-Constraint-Solvers durch intelligente Suchverfahren an (z. B. durch eine Backjumping-Variante, vgl. Abschnitt 5.2.4.3, S. 117).

Die Effizienz der globalen Lösungssuche kann zudem gesteigert werden, wenn weitere bestehende Inkonsistenzen vor der Initiierung des Suchvorgangs durch Ausnutzung des in den Teillösungen vorhandenen Wissens eliminiert werden. Bezogen auf ein spezifisches Problem stellt die Aufzählung aller möglichen Lösungen globale Konsistenz hinsichtlich der Wertebereiche dar. Für die vorhandenen Teillösungen bedeutet dies, dass sich die Wertebereiche der Constraint-Variablen ggf. weiter einschränken lassen. Dies resultiert daraus, dass sich mögliche Teillösungen nicht addieren, sondern sich gegenseitig ausschließen und ggf. Inkonsistenzen identifizieren: Werte, die nicht innerhalb einer Teillösung als gültige Belegung auftauchen, sind als inkonsistent herausgefiltert worden und dürfen dementsprechend bei Überlappungen der Constraint-Netze auch in keiner anderen Teillösung respektive einer globalen Lösung vorkommen. Entsprechend können aus Teillösungen eingeschränkte Wertebereiche abgeleitet werden. Ein Algorithmus muss folgendermaßen vorgehen:

1. Die ursprünglichen Wertebereiche der Variablen werden in einer temporären Struktur zwischengespeichert.

[10] Anstatt einer n-dimensionalen Matrix kann eine Teillösung zur Vereinfachung, und analog zur Repräsentation von extensionalen (Tupel-)Constraints in EngCon, als Tabelle mit der Anzahl Spalten aufgefasst werden, wie Variablen im jeweiligen Teilproblem existieren, und mit der Anzahl Zeilen, wie gültige Relationen in Form von einzelnen Teillösungen existieren.

Gegeben:

- $P_{Meta} = \{P_1, \ldots, P_m\}$, Meta CSP mit unterschiedlichen Teilproblemen.
- $V = \{v_1, \ldots, v_n\}$, eine endliche Menge von Variablen mit assoziierten Wertebereichen $D = \{D_1, \ldots, D_n\}$ mit $\{v_1 : D_1, \ldots, v_n : D_n\}$.
- $T = \{T_1, \ldots, T_m\}$, Mengen von Teillösungen der Teilprobleme $P_1, \ldots, P_m$, wobei jede Teillösung in T_i, $i \in \{1, \ldots, m\}$, eine Teilmenge der Variablen $v_j \in V$, $j \in \{1, \ldots, n\}$ mit einem jeweils zugeordnetem Wert $d_j \in D_j$ beinhaltet.

Algorithmus:

0. $D_{copy} \leftarrow D$.
1. **for each** P_i, $i \in \{1, \ldots, m\}$, **do**
   (a) **for each** v_j aus P_i **do**
      i. Entferne alle Teillösungen in $\{T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_m\}$, die Wertebelegungen d_j für v_j enthalten, die nicht in den aktuellen Teillösungen T_i aber in $D_{copy,j}$ enthalten sind.
      ii. Entferne alle Wertebelegungen d_j aus $D_{copy,j}$, wenn d_j nicht in einer Teillösung aus der aktuellen Menge von Teillösungen T_i enthalten ist.

Abbildung 6.10: Algorithmus zur Unterstützung eines Meta-Constraint-Solvers.

2. Für alle Teilprobleme und jeweils für alle Variablen v des aktuellen Teilproblems gilt: Alle Werte für v, die im Vergleich zur kopierten Domäne nicht in der aktuellen Teillösung enthalten sind, können ebenfalls nicht Teil einer Lösung in anderen Teillösungen sein. Die entsprechenden, einzelnen Teillösungen (anderer Constraint-Lösungsstrategien bzw. Teilprobleme), die trotzdem eine derartige Belegung für v enthalten, sind zu entfernen.[11]

3. Zur Vermeidung redundanter Such- und Vergleichsvorgänge sind jeweils Wertebelegungen, nach denen bereits gesucht worden ist, aus den zwischengespeicherten ursprünglichen Wertebereichen zu entfernen. Anschließend erfolgt die Überprüfung der nächsten Wertebelegung, bis sämtliche Variablen in allen Teilproblemen überprüft worden sind.

Ein Algorithmus, der zur Unterstützung eines Meta-Constraint-Solvers aufgrund der vorhandenen Teillösungen inkonsistente Teillösungen entfernt, ist in Abbildung 6.10 zu sehen. Durch die Vorgehensweise des Algorithmus wird kein spezifischer Konsistenzgrad hergestellt, da keine Wertekombinationen verglichen und auf Konsistenz überprüft werden. Durch einen Algorithmus, der oben beschriebene (Teil-)Relationen entfernt, werden lediglich die Wertebereiche der Variablen der einzelnen Teillösungen angeglichen – in diesem Fall anhand extensional repräsentierter Relationen.

[11] Entspricht dem Löschen einer Zeile aus der Tabelle mit der Menge der Teillösungen für ein Teilproblem.

Ebenso können in Bezug auf das Meta CSP in weiteren Algorithmen Konsistenzverfahren zur Vereinfachung eingesetzt werden. Die Anwendung geht allerdings mit einer hohen Komplexität einher, da das Meta CSP durch n-äre, extensionale Relationen – den (vollständigen) Teillösungen – repräsentiert wird. Die Komplexität des Problems ist abhängig davon, wieviele Variablen an den einzelnen Teillösungen beteiligt sind. Eine Teillösung, die z. B. zwanzig Variablen umfasst, würde entsprechend die Behandlung eines 20-stelligen Constraints bedeuten.

In Bezug auf Konsistenz- und Suchverfahren, die auf das Meta CSP angewendet werden, ist weiterhin zu beachten, dass ggf. heterogene Constraints innerhalb eines heterogenen Constraint-Problems verarbeitet werden müssen. Dies sind Constraints, die sowohl Variablen mit finiten als auch infiniten Wertebereichen gleichzeitig beschränken. Die entsprechenden Konvertierungsmethoden, d. h. Diskretisierungen und Überführungen von finiten Wertebereichen in kontinuierliche Intervalldomänen, sind zu berücksichtigen und die entsprechenden Lösungsalgorithmen anzupassen.

Grundsätzlich problematisch in Bezug auf ein Meta CSP sind Lösungsverfahren, die unvollständig sind. Dies sind Suchverfahren, die nicht alle Lösungen eines (Teil-)Problems berechnen, sondern z. B. aus Effizienzgründen lediglich die Erstbeste. Werden solche Lösungsverfahren eingesetzt, so kann auch insgesamt die Vollständigkeit nicht garantiert werden, d. h. es können mögliche Lösungen für ein Problem verloren gehen. Bezogen auf einen Meta-Constraint-Solver bedeutet dies, dass es äußerst unwahrscheinlich werden kann eine globale Lösung zu finden, wenn für jedes Teilproblem max. eine einzige Teillösung generiert wird.

Weiterhin ist zu beachten, dass in allen Phasen des Lösungsvorgangs und in allen Strategien die Inkrementalität der Lösungsverfahren gewährleistet sein muss. Das heißt, dass bei einer erneuten Propagation mit zusätzlichen Constraints und Variablen die vormals erstellten Constraint-Netze nicht erneut instantiiert werden müssen. In Bezug auf Preprozessingverfahren sind hinzukommende Variablen und Constraints unkritisch. Das Hinzufügen neuer (umfassender) Variablen für eine Binärisierung von FD-Constraints ist ebenso wie das Hinzufügen neuer Variablen bei einer Zerlegung von (Intervall-)Constraints in primitive Constraints voneinander unabhängig in jedem Teilproblem inkrementell möglich. Auf Konsistenz- und Suchverfahren trifft dies ebenfalls zu. Die anwachsenden Constraint-Netze müssen innerhalb der Strategien von einer Phase zur nächsten „durchgereicht“ werden. Ob Inkrementalität dabei von den einzelnen Constraint-Verfahren selbst angeboten wird, oder ob dies ggf. eine geeignete (Wrapper-)Schnittstelle simuliert (z. B. für eingebundene Fremdsysteme), ist für YACS unerheblich. Einzelne Constraint-Solver werden als *Black Box* aufgefasst. Die im Rahmen dieser Arbeit umzusetzenden Constraint-Lösungsverfahren sollten allerdings inkrementell anwachsende Constraint-Netze von sich aus unterstützen.

6.5 Systemarchitektur von YACS

Aufgrund einer Vielzahl von Constraint-Lösungsverfahren und möglicher Kombinationen derselben, deren unterschiedlichen Eigenschaften und der problemabhängigen bzw. anwen-

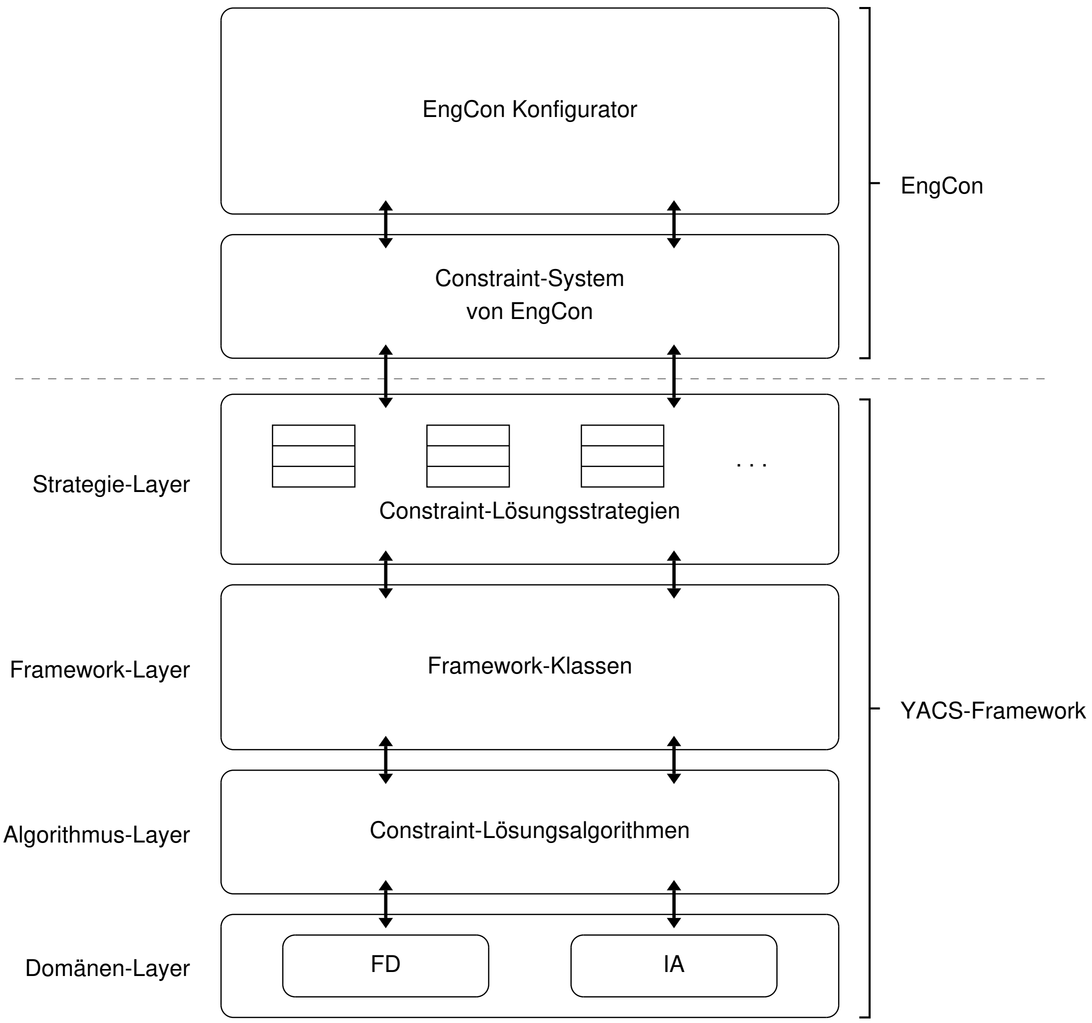


Abbildung 6.11: Systemarchitektur von YACS

dungsspezifischen Effizienz unterschiedlicher Verfahren, ist zur Unterstützung der strukturbasierten Konfigurierung eine Komponente notwendig, mit der sich flexibel, je nach Problemstellung unterschiedliche Constraint-Lösungsmechanismen einsetzen lassen. Das im Rahmen dieser Arbeit entwickelte YACS-Framework (vgl. Abbildung 6.11) stellt eine modulare und wiederverwendbare Constraint-Lösungskomponente dar. YACS ist ein hybrides System für den flexiblen Einsatz von Constraint-Lösungsverfahren für finite und infinite Domänen. Die Lösungsverfahren sind eingebettet innerhalb einer strategiebasierten, modularen Framework-Architektur:

- **Constraint-Lösungsstrategien**

  Der flexible Einsatz von Constraint-Lösungsverfahren wird über ein Strategiekonzept realisiert. Abstrahiert von den eigentlichen Lösungsalgorithmen können von dem Wissensingenieur problemabhängig bzw. anwendungsspezifisch unterschiedliche

Constraint-Lösungsstrategien eingesetzt werden. Diese Lösungsstrategien müssen vorab in Abhängigkeit von den vorhandenen Lösungsverfahren definiert werden.

- **Framework-Architektur**

  Durch die Framework-Architektur wird sichergestellt, dass Lösungsverfahren flexibel ausgetauscht und zudem auf einfache Weise neue Lösungsalgorithmen implementiert bzw. Fremdsysteme integriert werden können. Das YACS-Framework stellt hierfür einen geeigneten Rahmen mit einheitlichen Schnittstellen bereit.

In Abbildung 6.11 auf der vorherigen Seite ist eine Übersicht über die Systemarchitektur von YACS, angebunden an das Konfigurierungswerkzeug EngCon, zu sehen. Aufsetzend auf einem **Domänen-Layer**, einer Umgebung zur arithmetischen Verarbeitung von finiten Domänen (FD) und reellwertigen Intervallen (eine Intervallarithmetik, kurz IA), werden die eigentlichen Algorithmen zum Auflösen von Constraint-Problemen implementiert (**Algorithmus-Layer**). Constraint-Verfahren aus Fremdsystemen können an dieser Stelle über Wrapper-Klassen eingebunden werden. Die Algorithmen bzw. die sie umschließenden (Wrapper)-Klassen müssen wiederum den Schnittstellen des **Framework-Layers** von YACS genügen.

Der durch das Framework vereinheitlichte Zugriff auf Constraint-Lösungsverfahren ermöglicht es dem **Strategie-Layer**, dem Anwender bzw. dem übergeordnetem System (in diesem Fall das Constraint-System von EngCon) eine flexible Auswahl an Lösungsverfahren anbieten zu können.[12] Abstrahiert von den Lösungsverfahren können auf dieser Ebene aus einer Reihe vordefinierter Constraint-Lösungsstrategien problemabhängig die für die jeweilige Anwendung geeigneten Strategien ausgewählt werden.

6.6 Diskussion

Das Framework-Konzept von YACS bietet eine flexible und nutzerfreundliche Architektur zur Implementierung von Constraint-Lösungsverfahren (vgl. Abschnitt 6.2, S. 177 f.). Die Framework-Architektur wiederum wird in Bezug auf die Abstraktion von den tatsächlich eingesetzten Verfahren optimal ergänzt durch das in Abschnitt 6.3 auf Seite 179 ff. beschriebene Strategiekonzept.

In den Constraint-Lösungsstrategien werden die tatsächlich einzusetzenden Lösungsverfahren definiert. Dies geschieht problemabhängig und flexibel je nach Anwendung und Einsatzzweck. Der Wissensingenieur kann sich somit bei der Erstellung der Wissensbasis auf vordefinierte und dokumentierte Constraint-Lösungsstrategien stützen, und diese zur Behandlung der im Rahmen der Konfigurierung entstehenden Constraint-Probleme gezielt einsetzen. Sollten Anpassungen notwendig werden, so ist eine einfache Wartung, Pflege und auch Neuimplementierung oder -anbindung von Constraint-Lösungsverfahren durch eine modulare Struktur und die Framework-Architektur gewährleistet.

[12] Auf eine detaillierte Darstellung der weiteren Systembestandteile von EngCon wurde in Abbildung 6.11 aus Gründen der Übersichtlichkeit verzichtet. Für eine vollständige Übersicht über die Architektur von EngCon siehe Abbildung 3.1 auf Seite 22.

Die Unterstützung finiter und infiniter Domänen ist sichergestellt durch die Nutzung eines hybriden CSPs, welches die beschriebenen Constraint-Lösungsstrategien beinhaltet. Lösungsstrategien können sowohl Constraint-Lösungsverfahren für finite als auch infinite Domänen enthalten. Durch die Eigenentwicklung der Framework-Architektur und der Constraint-Lösungsverfahren wird sichergestellt, dass sowohl die Schnittstellen als auch die Verfahren selber den inkrementellen Aufbau des Constraint-Netzes zu unterstützen.

In Abschnitt 6.4.1 auf Seite 182 ff. wurden in zwei Szenarien unterschiedliche Ausführungsmodelle dargelegt. In Verbindung mit den in Abschnitt 6.4.2 auf Seite 186 ff. und Abschnitt 6.4.3 auf Seite 191 ff. diskutierten Aspekten bzgl. Überlappungen von Constraint-Netzen und heterogenem Constraint-Lösen ergeben sich insgesamt vier Möglichkeiten hinsichtlich der Funktionalität einer Realisierung von YACS:

1. **Zwei Strategien und getrennte Constraint-Netze (lokale Sichtweise):**

   Es werden für jedes Konfigurierungsproblem ausschließlich zwei Constraint-Lösungsstrategien – je eine für finite und infinite Domänen – vorgesehen. Die Constraint-Netze der beiden Teilprobleme dürfen sich nicht überlappen.

   Diese Voraussetzung resultiert in zwei voneinander disjunkten Constraint-Netzen, die unabhängig voneinander propagiert und aufgelöst werden können, ähnlich wie zurzeit innerhalb von EngCon die unterschiedlichen Constraint-Arten (Tupel-, Java- und Funktions-/Prädikat-Constraints) separat behandelt werden. Die „Metapropagation“ zur Erstellung von konsistenten Konfigurierungslösungen wird in diesem Fall von EngCon geleistet.

2. **Beliebig viele Strategien und getrennte Constraint-Netze (lokale Sichtweise):**

   Diese Annahme dehnt obiges auf beliebig viele Constraint-Netze aus, die sich ebenfalls nicht überschneiden dürfen. Für jedes Konfigurierungsproblem können beliebig viele Strategien definiert und eingesetzt werden (theoretisch für jedes Constraint eine andere). Die Constraint-Netze der jeweils entstehenden Teilprobleme dürfen sich allerdings auch hier aus Gründen der Vereinfachung nicht überlappen.

   Jedes Teilproblem wird damit separat propagiert, ebenso wie die unterschiedlichen Constraint-Arten (Tupel-Constraints, Java-Constraints, etc.) innerhalb von EngCon. Auch in diesem Fall kommt EngCon die Aufgabe zu, die einzelnen Teillösungen innerhalb einer konsistenten Gesamtkonfiguration zusammenzuführen.

3. **Zwei Strategien und überlappende Constraint-Netze (globale Sichtweise):**

   In diesem Fall werden wiederum lediglich zwei Strategien (finite und infinite Domänen) für ein Konfigurierungsproblem zugelassen. Da Überlappungen möglich sein sollen, ist ein Meta-Constraint-Solver erforderlich, der aufgrund der begrenzten Anzahl unterschiedlicher Teilprobleme relativ überschaubaren Verwaltungsaufwand zu leisten hat.

   Trotzdem müssen bei einer Realisierung dieser Möglichkeit die in Abschnitt 6.4.1 auf Seite 182 ff. und insbesondere Abschnitt 6.4.3 auf Seite 191 ff. beschriebenen

Methoden bzgl. überlappender Constraint-Netze und zum Auflösen des entstehenden heterogenen Constraint-Problems umgesetzt werden. Dadurch wird YACS in die Lage versetzt, globale Lösungen für die zu verarbeitenden algebraischen Constraints zu generieren.

4. **Beliebig viele Strategien und überlappende Constraint-Netze (globale Sichtweise):**

   Diese Möglichkeit bietet maximale Flexibilität hinsichtlich des Einsatzes unterschiedlicher Constraint-Lösungsstrategien und in Bezug auf das in Abschnitt 6.4.1 auf Seite 182 ff. geschilderte, phasenweise Ausführungsmodell. Es können beliebig viele Strategien zur Verarbeitung unterschiedlicher Teilprobleme genutzt werden, wobei wiederum Überlappungen der Constraint-Netze erlaubt sind. Es ist daher wie bei der vorhergehenden Möglichkeit ein Meta-Constraint-Solver erforderlich. In diesem Fall allerdings wird ein generischer Ansatz benötigt, d. h. für beliebig viele Teilprobleme.

   Hierbei ist es insbesondere erforderlich, die in Abschnitt 6.4.1 auf Seite 182 ff. und Abschnitt 6.4.3 auf Seite 191 ff. angesprochenen Aspekte und Lösungsmöglichkeiten bzgl. heterogener Constraint-Probleme und überlappender Constraint-Netze zu berücksichtigen. YACS wird es dadurch ermöglicht, globale Lösungen für das vorliegende heterogene Constraint-Problem, bestehend aus unterschiedlichen Teilproblemen, zu erzeugen.

Im Rahmen dieser Arbeit soll wenigstens eines dieser vier Szenarien umgesetzt werden. Je flexibler das realisierte Szenario ist, umso eher besteht die Möglichkeit, dass YACS in EngCon und anderen Projekten tatsächlich eingesetzt wird. Wenn sich auch das erstgenannte Szenario mit lediglich zwei voneinander unabhängigen Constraint-Netzen am einfachsten realisieren lässt, würden hier weitergehende Möglichkeiten zur Flexibilisierung des Einsatzes von Constraint-Lösungsverfahren nicht wahrgenommen. Trotzdem lassen sich auch in diesem einfachen Szenario bereits die Anforderungen für eine Anwendung innerhalb der strukturbasierten Konfigurierung mit EngCon umsetzen und deutliche Verbesserungen des derzeitigen Zustands hinsichtlich des flexiblen Einsatzes unterschiedlicher Constraint-Lösungsverfahren erreichen.

Kapitel 7

Realisierung und Anbindung an EngCon

My software never has bugs,
it just develops random features.

ANONYM

Dieser Abschnitt enthält die Beschreibung der Implementierung des YACS-Frameworks. Es werden der Reihe nach die enthaltenen Komponenten aufgezählt und dokumentiert. Abschließend erfolgt die Beschreibung der Integration von YACS in das Konfigurierungswerkzeug ENGCON.

7.1 Einleitung

Die Implementierung von YACS erfolgt, bedingt durch den Anwendungsfall ENGCON, unter der Windows-Plattform. Durch die Programmiersprache Java ist allerdings eine weitestgehende Plattformunabhängigkeit gegeben (vgl. Middendorf et al. 2002). Das verwendete Java von „Sun Microsystems" ist die Entwicklerversion der *Java 2 Platform, Standard Edition* in der Version 1.4.2 (J2SE 1.4.2 SDK). Selbiges ist in der genutzten Entwicklungsumgebung „Borland JBuilder 2005 Foundation" enthalten.

Die benötigte intervallarithmetische Funktionalität liefert die Bibliothek IAMath[1] von Timothy J. Hickey (vgl. Abschnitt 4.5.2, S. 72). Intervallarithmetische Operationen werden standardmäßig von den wenigsten Programmiersprachen unterstützt. Die frei verfügbare IAMath-Bibliothek ermöglicht grundlegende intervallarithmetische Operationen und Funktionen und ist ebenso wie ENGCON vollständig in Java implementiert. Um YACS nicht auf eine bestimmte Bibliothek festzulegen, wurde im Entwurf eine entsprechende Kapselung vorgesehen.

[1] `http://interval.sourceforge.net/interval/java/ia_math/README.html`

Constraint-Lösungsstrategien werden für das YACS-Framework innerhalb einer XML-Datei definiert (vgl. Eckstein 2000). Der dafür notwendige XML-Parser von YACS ist in der in Java enthaltenen *Java API for XML Processing* (JAXP)[2] implementiert (vgl. Armstrong et al. 2005, S. 109 ff.) und nutzt das standardmäßig integrierte *Document Object Model* (DOM) von Sun (vgl. Armstrong et al. 2005, S. 181 ff.). Dies hat den Vorteil, dass keine separate XML-Bibliothek benötigt wird.

Der stringbasierte Parser von YACS für das Einlesen von Constraint-Ausdrücken wird über eine JLex/Java CUP Kombination realisiert. Während JLex[3] für die lexikalische Analyse basierend auf regulären Ausdrücken zuständig ist, steht mit Java CUP[4] ein Parsergenerator aufbauend auf den Eingaben von JLex zur Verfügung (vgl. Berk 2000; Hudson 1999). Beide Programme sind Open-Source und daher frei verfügbar.

Als Logging-Mechanismus kommt Log4J[5] zum Einsatz. Der Open-Source Logger der „Apache Foundation" ist leistungsfähiger als das im Standard-Java integrierte Logging und hat im Gegensatz dazu eine hohe Verbreitung gefunden (vgl. Gülcü 2002; Schnelle 2004). Dies begünstigt die Integration in das YACS-Framework, welches somit neben einer Integration in EngCon auch für eine große Zahl anderer Systeme eine möglichst hohe Kompatibilität diesbezüglich bietet.

7.2 Das Framework-Konzept

Der objektorientierte Framework-Entwurf zählt zu den anspruchsvollsten Entwurfsaufgaben. Im Gegensatz zu (Klassen-)Bibliotheken, für die eine „Codewiederverwendung" im Vordergrund steht, muss für ein Framework die „Entwurfswiederverwendung" für bestimmte Klassen von Software gewährleistet sein (vgl. Gamma et al. 1996, S. 37).

Im Falle von YACS stellt sich dies übersichtlich dar: Neben einer einfachen Benutzung durch den Anwender bzw. Wissensingenieur auf der einen Seite, muss es innerhalb von YACS möglich sein, neue Lösungskomponenten für Constraint-Probleme zu entwickeln. Die Schnittstelle hierfür ist, wie im Folgenden ersichtlich wird, denkbar schlank gehalten. Jeder Constraint-Solver muss eine abstrakte Solver-Klasse mit einer aufrufenden Methode implementieren. Dieser Methode werden zur Laufzeit die benötigten Informationen zu einem bestimmten Teilproblem übergeben. Der Constraint-Solver arbeitet als *Black Box* und liefert seine Ergebnisse, je nach Kategorie (Preprozessing, Konsistenzherstellung, Lösungssuche), an den aufrufenden Constraint-Manager bzw. an das Teilproblem.

Die Domäneninformationen zu einem bestimmten Problem sind wiederum durch generische (abstrakte) Framework-Klassen gekapselt. Das YACS-Framework ist damit sowohl in Bezug auf die Constraint-Solver als auch auf die zu verarbeitenden Domänen erweiterbar. Neben reellwertigen Intervalldomänen sind Domänenklassen für finite Domänen mit Integer-Werten und für symbolische, d. h. stringbasierte, Domänen implementiert.[6] Dabei

[2] `http://java.sun.com/webservices/jaxp/docs.html`

[3] `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

[4] `http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[5] `http://logging.apache.org/log4j/`

[6] Denkbare Erweiterungen der mit dem YACS-Framework verarbeitbaren Wertebereiche könnten z. B. spezielle Implementierungen für boolesche oder reellwertige Domänen sein. Beide Domänen lassen sich der-

ist festzuhalten, dass sich durch die implementierten FD-Solver aufgrund der Kapselung durch die Framework-Klassen, sowohl numerische als auch symbolische Domänen verarbeiten lassen.

Durch YACS vorgegeben ist neben den zu implementierenden Klassen lediglich die Ausführungskontrolle. Dies entspricht ebenfalls der klassischen Definition eines Frameworks (vgl. Gamma et al. 1996, S. 37). In Bezug auf YACS bezeichnet Ausführungskontrolle die Steuerung des phasenweisen Lösungsprozesses, während dessen die in den Strategien definierten Lösungsverfahren auf die jeweiligen Teilprobleme angewendet werden (vgl. Abschnitt 6.4.1, S. 182).

Im Folgenden wird der Softwareentwurf für das YACS-Framework vorgestellt. Zur Dokumentation werden Diagramme in der Modellierungssprache UML verwendet (vgl. Fowler und Scott 2000).[7] Für die Implementierung von YACS wurde auf eine Reihe von *Design Patterns* bzw. „Entwurfsmuster“ zurückgegriffen (vgl. Gamma et al. 1996), auf die an den jeweiligen Stellen eingegangen wird. Eine ausführliche Beschreibung der angesprochenen Klassen und insbesondere der darin enthaltenen Methoden befindet sich in der API-Dokumentation in Anhang E auf Seite 285 ff.

7.3 Übersicht über die Packages

Neben einem übergeordnetem Package `yacs` mit dem darin enthaltenen Constraint-Manager von YACS, existieren die folgenden Unterpackages (siehe Abbildung 7.1 auf der nächsten Seite):

- `yacs.parser`

  Hierin sind der JLex/Java-CUP-Parser zum Einlesen von stringbasierten Constraint-Ausdrücken sowie die benötigten Klassen zur Repräsentation von Constraint-Ausdrücken als Objektstruktur enthalten.

- `yacs.domain`

zeit bereits verarbeiten: Boolesche-Constraints können als Sonderfall von Integer-Constraints, reellwertige Constraints entsprechend als Sonderfall von reellwertigen Intervall-Constraints (in diesem Fall Punktintervalle) gesehen werden. Eine explizite Implementierung dieser Domänen ist allerdings ebenso möglich.

[7] Eine Besonderheit der in dieser Arbeit verwendeten UML-Diagramme liegt in Bezug auf die Darstellung von Klassendiagrammen vor. Entgegen der üblichen Konvention zur Darstellung von UML-Klassendiagrammen, die ausschließlich Attribute und Operationen kennt (vgl. Fowler und Scott 2000), sind die Diagramme in dieser Arbeit um ein drittes Feld, den „Eigenschaften“ einer Klasse, ergänzt. Eine Eigenschaft liegt immer dann vor, wenn einem Methodennamen, der einem Attributnamen entspricht, „is“, „get“ oder „set“ vorausgeht. Der Attributname `value` bspw. mit der Methode `getValue()` ist eine Eigenschaft. Eigenschaften haben die Sichtbarkeit der jeweiligen *Getter-/Setter*-Methoden. Die Notation der Klassen und Interfaces, die derartige Eigenschaften aufweisen, wird außerdem um ein kleines Rechteck am linken Rand ergänzt. Diese Art der Darstellung ist, bedingt durch die Verwendung der Komponente zur Generierung von UML-Diagrammen „Borland Together Developer 2005“ in Kombination mit der frei verfügbaren Entwicklungsumgebung JBuilder *Foundation*, fest vorgegeben und kann ausschließlich in der *Enterprise*-Edition der IDE beeinflusst werden.

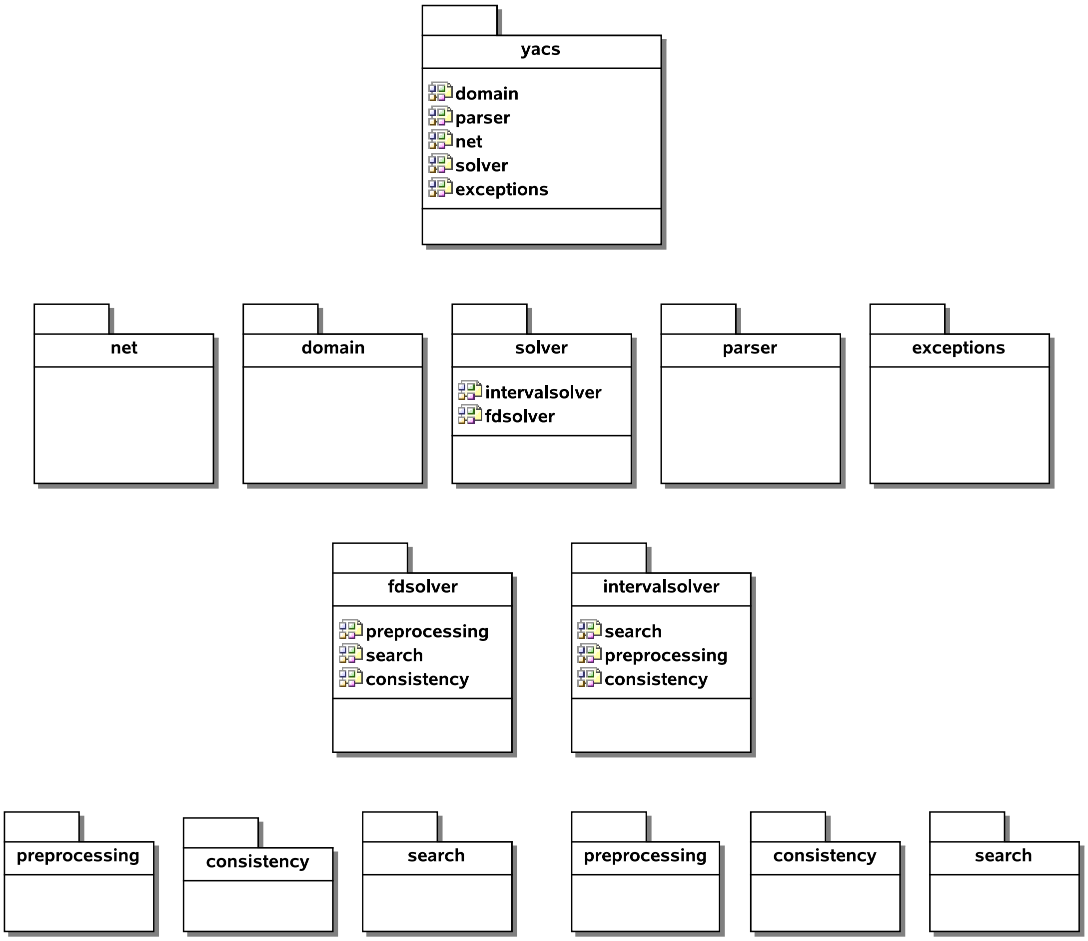


Abbildung 7.1: Übersicht über die Package-Struktur von YACS

Enthält die benötigten und z. T. abstrakten Klassen zur Repräsentation und Kapselung unterschiedlicher Wertebereiche, die von Constraint-Solvern innerhalb von YACS verarbeitet werden.

- `yacs.solver`

  In diesem Package sind abstrakte Klassen enthalten, die von Constraint-Solvern implementiert werden müssen, die in das YACS-Framework eingebunden werden sollen. Außerdem sind eine Reihe von Unterpackages enthalten, in denen die konkreten Solver-Klassen abgelegt werden können.

- `yacs.net`

  Enthält Klassen zur Repräsentation der Constraint-Netze und der zugehörigen Strategien sowie den benötigten XML-Parser zum Einlesen von Constraint-Lösungsstrategien.

- `yacs.exceptions`

  Für die Fehlerbehandlung wurden eine Reihe von *Exceptions* vordefiniert, die von zu implementierenden Constraint-Solvern bzw. von den bereits existierenden Komponenten von YACS genutzt werden können, um Ausnahmesituationen anzuzeigen.

Das Package `yacs.solver` beinhaltet neben abstrakten Solver-Klassen die Unterpackages `yacs.solver.fdsolver` und `yacs.solver.intervalsolver`, in die entsprechend Constraint-Lösungsverfahren für finite und infinite (Interval-)Domänen abgelegt werden. Die Package-Struktur untergliedert sich dafür an dieser Stelle noch einmal: Sowohl in `fdsolver` als auch in `intervalsolver` sind die folgenden Unterpackages enthalten:

- `preprocessing`
- `consistency`
- `search`

In diese Packages werden die Constraint-Solver, je nach Kategorie und inkl. der ggf. benötigten Hilfsklassen, abgelegt.

7.4 Der Constraint-Manager

Der YACS Constraint-Manager (YCM) dient als zentrales Bindeglied zwischen einer Anwendung und dem YACS-Framework. Der YCM vereint die benötigten Methoden zur Definition von Constraint-Problemen, zur Initiierung der Auswertung und für das Auslesen der Ergebnisse. Diese Funktionalität entspricht dem Entwurfsmuster einer „Fassadenklasse“ (engl. *facade*): Der YCM verbirgt komplexere Schnittstellen vor dem Nutzer des YACS-Frameworks (vgl. Gamma et al. 1996, S. 212 ff.).

Außerdem werden im YCM die Constraint-Lösungsstrategien bzw. die ihnen zugeordneten Constraint-Netze verwaltet und der Lösungsprozess gesteuert. Der YCM ist somit ebenfalls Bindeglied zwischen den internen Komponenten des YACS-Frameworks.

Die Realisierung des Constraint-Manager unterteilt sich in eine abstrakte Interface-Klasse, die sämtliche benötigten Methoden definiert, und einer Implementierung dieser Klasse (siehe Abbildung 7.2 auf der nächsten Seite).[8] Durch diese Abstraktion ist es möglich, ggf. unterschiedliche Implementierungen des YCM zu verwenden, die sich z. B. hinsichtlich der Ausführungskontrolle des Lösungsprozesses unterscheiden können. Es muss allerdings gewährleistet sein, dass diese Realisierungen ebenfalls das Interface `YacsConstraintManager` implementieren.

Die Implementierung `YacsConstraintManagerImpl` wird mit dem Dateipfad zu einer XML-Datei mit den Constraint-Lösungsstrategien instantiiert. Diese Lösungsstrategien werden ausgelesen und für jede Strategie wird ein (leeres) Constraint-Netz erzeugt.

[8] Attribute, Operationen und Eigenschaften einer Klasse, die diese aufgrund einer Generalisierung von einer übergeordneten Klasse geerbt hat, werden aus Gründen der Übersichtlichkeit in diesem und den nachfolgenden Klassendiagrammen nach Möglichkeit nicht redundant aufgeführt.

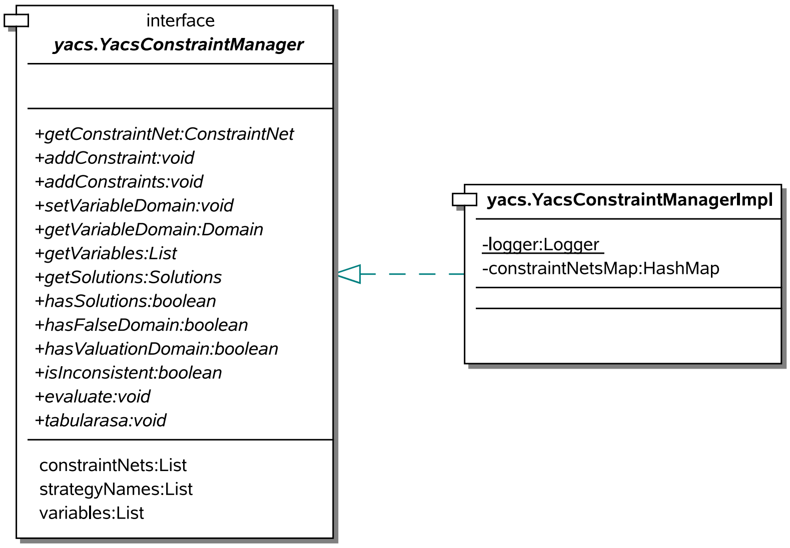


Abbildung 7.2: Der YACS Constraint-Manager (YCM)

Constraints bzw. Constraint-Teilprobleme lassen sich über den Constraint-Manager inkrementell zu Constraint-Netzen hinzufügen. Die Constraint-Netze werden anhand der Namen der ihnen zugehörigen Constraint-Lösungsstrategien unterschieden. Bei dem Hinzufügen eines Constraint-Ausdrucks muss dem YCM mitgeteilt werden, welcher Constraint-Lösungsstrategie bzw. welchem Constraint-Netz der jeweilige Ausdruck zugeordnet werden soll.

7.5 Repräsentation von Constraint-Ausdrücken

Wie bereits in der Einleitung erwähnt, wird die stringbasierte Constraint-Schnittstelle von YACS über eine Kombination von JLex und dem Java-CUP-Parsergenerator erreicht. Mit JLex wird eine lexikalische Analyse auf den übergebenen Strings durchgeführt (vgl. Berk 2000). Darauf aufbauend können anschließend durch Java CUP anhand einer definierten Parser-Grammatik syntaktisch korrekte Ausdrücke in eine Objektstruktur überführt werden (vgl. Hudson 1999).

JLex und Java CUP orientieren sich an den klassischen Werkzeugen Lex und Yacc (vgl. Johnson 1975; Lesk und Schmidt 1975) bzw. den freien Alternativen Flex und Bison (vgl. Donnelly und Stallman 2002; Paxson 1995) zur Parsergenerierung: Während einer vorgeschalteten lexikalischen Analyse erfolgt durch einen „Scanner“ (auch „Lexer“ genannt) auf Basis von regulären Ausdrücken die Zerlegung einer Eingabe in eine Folge von logisch zusammengehöriger Einheiten, in sogenannte „Token“[9]. Der nachfolgende Parser nutzt die vom Scanner erkannten Token als „Terminalsymbole“[10] zur synaktischen

[9] „Token“, von engl. *tokens*: Zeichen, Kürzel, Marken

[10] Atomare Symbole, die nicht weiter zerlegt werden können. Terminalsymbole kommen entsprechend *nicht* auf der linken Seite einer Regel innerhalb einer formalen Grammatik vor.

Analyse anhand einer Parser-Grammatik in Backus-Naur-Form (BNF). Mit diesen, aus dem Compilerbau stammenden Werkzeugen, ist die Verarbeitung kontextfreier Grammatiken, z. B. für Programmiersprachen, möglich (vgl. Herold 1999, S. 277 ff.). Mit den Java-Pendants JLex/Java CUP lässt sich entsprechend auf einfache Weise ein Parser für die in diesem Fall benötigten algebraischen Ausdrücke realisieren.

Der Constraint-Parser von YACS ist derzeit in der Lage, einfache algebraische Gleichungen und Ungleichungen zu interpretieren. Neben der Eingabedatei für den Scanner JLex ist eine CUP-Parser-Grammatik notwendig, welche beide in Anhang D auf Seite 281 ff. dokumentiert sind. Neben einzelnen Constraint-Ausdrücken ist der Parser entsprechend der spezifizierten Grammatik in der Lage, Aneinanderreihungen von primitiven Constraints zu interpretieren. Diese Möglichkeit, mehrere Constraints gleichzeitig zu einem Constraint-Problem hinzuzufügen, wurde aus Effizienzgründen vorgesehen. Falls die Anwendung kein inkrementell anwachsendes Constraint-Netz aufweist oder während eines Konfigurierungsschrittes mehrere neue Constraints vorliegen, können auf diesem Weg ganze Teilprobleme auf einmal geparst und zu einem Constraint-Netz hinzugefügt werden. Dies bedeutet weniger Overhead durch YACS, wie er beim Einfügen von einzelnen Constraints entstehen würde.

Eine Besonderheit ist die Fähigkeit des Parsers, für Variablen mit gleichem Namen ein und dasselbe Objekt zu verwenden. In Zeile 16 der Parser-Grammatik in Anhang D wird dafür eine `HashMap` instantiiert, die dazu dient, einmal erzeugte Variablen ggf. „wiederzufinden“ und dieselbe Objektreferenz erneut zu verwenden (Zeile 115–123). Um auch Objektreferenzen von Variablen verfügbar zu haben, die von einer anderen, vorhergehenden Parser-Instanz generiert wurden (die Constraint-Netze von YACS können inkrementell erzeugt werden), bietet der Parser die Möglichkeit, zwischen Instantiierung und Parsen eine `HashMap` zu übergeben (Methode `addVariablesMap()`, Zeile 35–37), mit welcher der eigentliche Parser zur Laufzeit rechtzeitig initialisiert wird (Zeile 54).

Auf diesem Weg wird ein globaler Namensraum bzw. eine globale Sicht innerhalb von YACS erreicht. Jede Variable ist unter ihrem eindeutigen Namen innerhalb eines jeden Constraint-Netzes ein und dasselbe Objekt. Die Wertebereichseinschränkung durch den Lösungsalgorithmus einer Strategie hat daher ggf. automatisch weitere Wertebereichseinschränkungen auch in anderen Strategien zur Folge, wenn dieselbe Variable in den jeweiligen Constraint-Netzen involviert ist, und dort eine Propagation der Wertebereiche initiiert wird.

Nachfolgend werden in diesem Abschnitt die Komponenten aufgezeigt, aus denen die vom Parser generierten Constraint-Ausdrücke bestehen. Sofern es sich um konkrete Ausprägungen handelt, sind sie Teil des Domänen-Layers von YACS. Abstrakte Klassen können entsprechend dem Framework-Layer zugeordnet werden (vgl. Abschnitt 6.5, S. 194).

7.5.1 Constraints

Der Constraint-Parser von YACS erzeugt Instanzen von Spezialisierungen der abstrakten Klasse `Expression`. Die Unterklassen `UnaryOperator` und `BinaryOperator` von `Expression` beinhalten selbst wieder eine bzw. zwei Instanzen eines Nachfolgers von `Expression`, wodurch eine rekursive Klassenstruktur zur Repräsentation von *Binärbäumen*

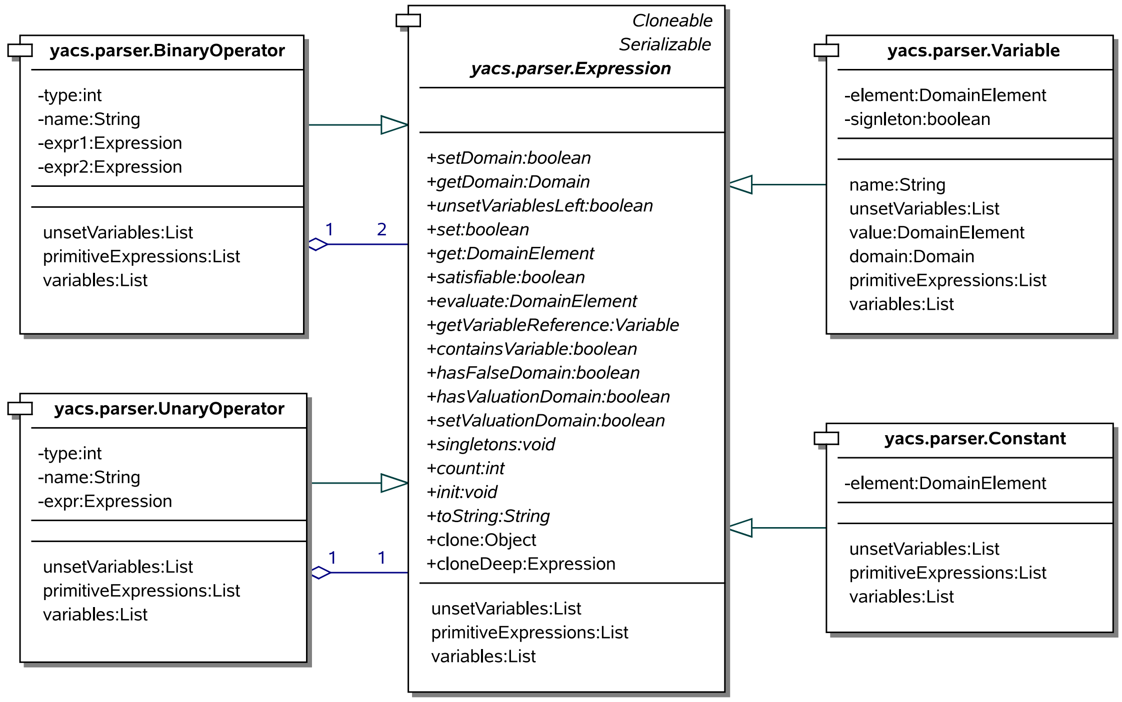


Abbildung 7.3: Repräsentation von Constraint-Ausdrücken durch Binärbäume

entsteht (siehe Abbildung 7.3). Die Klassenstruktur entspricht damit dem Entwurfsmuster „Kompositum" (engl. *composite*): Objekte lassen sich zu Baumstrukturen zusammenfügen, wobei sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich behandelt werden können (vgl. Gamma et al. 1996, S. 239 ff.).

Mit einem auf diese Weise definierten Binärbaum lässt sich ein primitiver Constraint-Ausdruck beschreiben. Die Klasse `BinaryOperator` stellt die Verzweigungen in unterschiedliche Äste durch binäre Operatoren dar ($=, \neq, \leq, \geq, <, >, +, -, *, /$). Durch die Klasse `UnaryOperator` werden einstellige Operatoren (derzeit ausschließlich Negation), durch `Constant` fixe Konstanten und durch die Klasse `Variable` entsprechend Variablen mit einer flexiblen Belegung und einer zugehörigen Wertedomäne repräsentiert.

Neben einzelnen, primitiven Constraints lassen sich mit Hilfe der Klasse `Expression` und deren Unterklassen ganze Teilprobleme durch eine Konjunktion von primitiven Constraints beschreiben. Der zusätzliche binäre Operator, der innerhalb des Baums eine solche Konjunktion beschreibt, ist das „;". Derart aneinandergehängte Constraint-Ausdrücke bzw. Constraint-Bäume werden innerhalb der Constraint-Netze von YACS genutzt, um das zugehörige Teilproblem zu repräsentieren.

Da manche Constraint-Lösungsalgorithmen die Duplizierung von Variablen, primitiven Constraints oder ganzen Teilproblemen voraussetzen (für echtes *call-by-value*), existiert in `Expression` die Methode `cloneDeep()`. Diese Methode gibt ein vollständig dupliziertes Objekt von seiner eigenen Instanz zurück. Im Gegensatz zur üblichen `clone()`-Methode werden sämtliche darin referenzierten Objekte ebenfalls dupliziert.

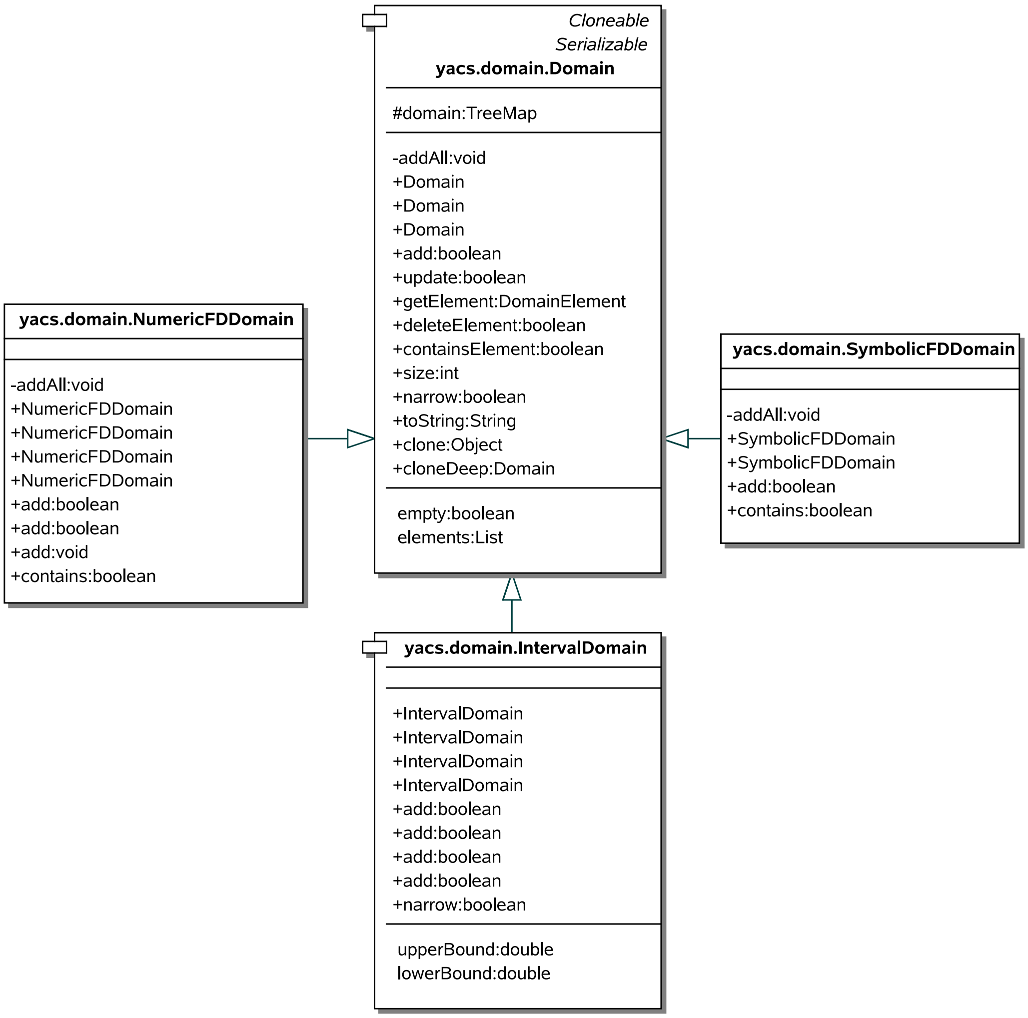


Abbildung 7.4: Wertebereiche von Constraint-Variablen

7.5.2 Wertebereiche

Die Wertebereiche von Constraint-Variablen werden durch die Klasse `Domain` repräsentiert. Die Klasse ist nicht abstrakt, d. h. vollständig implementiert, und kann für beliebige Wertebereiche genutzt werden, da eine entsprechende Kapselung der enthaltenen Elemente vorgesehen wurde (siehe Abbildung 7.4).

Aus Gründen der Vereinfachung beim Instantiieren bzw. beim Zugriff auf diese Klasse wurden Unterklassen von `Domain` erzeugt, die einen erleichterten Zugriff mit den jeweils entsprechenden Elementen der Domäne erlauben (numerische/symbolische finite Domänen, infinite Intervalldomänen). Sollen weitere Domänen unterstützt werden, ist das YACS-Framework an dieser Stelle durch weitere Unterklassen erweiterbar.

Instanzen der Klasse `Domain` bzw. deren Unterklassen müssen sich wie Instanzen der Klasse `Expression` vollständig duplizieren lassen, da sie Teil einer Variable sind (vgl. Ab-

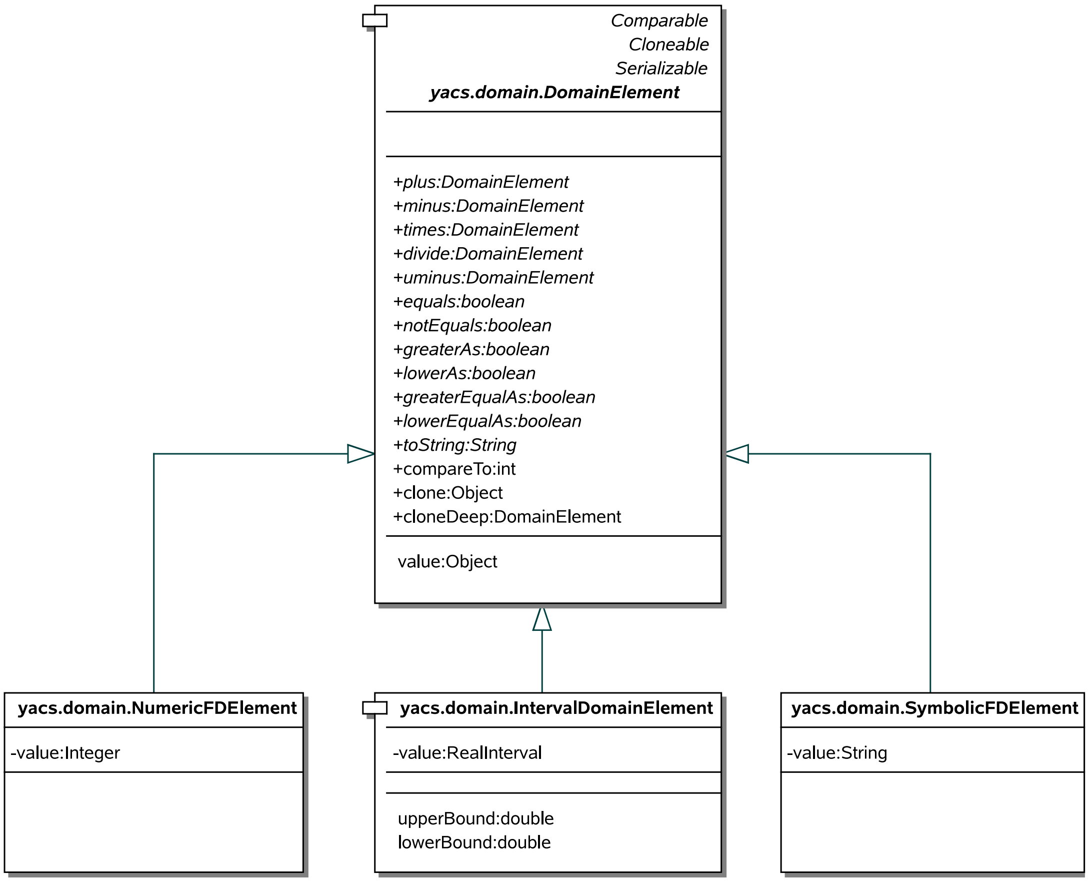


Abbildung 7.5: Kapselung der Elemente eines Wertebereichs

schnitt 7.5.1, S. 205). Die Klasse `Domain` verfügt daher ebenfalls über eine `cloneDeep()`-Methode.

Belegt werden die Wertebereiche mit Instanzen von Spezialisierungen der abstrakten Oberklasse `DomainElement`, mit denen sich Elemente aus unterschiedlichen Domänen kapseln lassen.

7.5.3 Elemente

In Abbildung 7.5 sind die abstrakte Klasse `DomainElement` und deren zurzeit existierenden Unterklassen zu sehen. Die Unterklassen von `DomainElement` kapseln jeweils die Elemente der Wertebereiche von Constraint-Variablen (Klasse `Domain`, vgl. Abschnitt 7.5.2 auf der vorherigen Seite). Die Klasse `DomainElement` und ihre Unterklassen entsprechen daher dem Entwurfsmuster „Adapter", welches auch unter dem Namen „Umwickler" (engl. *wrapper*) bekannt ist (vgl. Gamma et al. 1996, S. 171 ff.). Derzeit existieren drei Unterklassen von `DomainElement`:

- Die Klasse `NumericFDDomainElement` kapselt den Zugriff auf ein Integer-Objekt und repräsentiert damit ein Element für eine numerische finite Domäne.
- Durch die Klasse `SymbolicFDDomainElement` wird ein String-Objekt gekapselt. Sie repräsentiert ein Element einer symbolischen finiten Domäne.
- Von der Klasse `IntervalDomainElement` wird eine Instanz der Klasse `RealInterval` umschlossen. Objekte dieser Klasse werden mit Hilfe der IAMath-Bibliothek erzeugt und repräsentieren ein reellwertiges Intervall.

Da von YACS bisher ausschließlich konvexe Intervalle unterstützt werden, lässt sich eine Instanz der Klasse `IntervalDomain` (vgl. Abschnitt 7.5.2, S. 207) jeweils nur mit einem einzigen `IntervalDomainElement` belegen.

Im Gegensatz zur Klasse `Domain` muss die Klasse `DomainElement` abstrakt deklariert sein, da in ihr Berechnungs- und Vergleichsoperatoren für die Elemente der jeweiligen Domäne definiert werden, die von den Unterklassen entsprechend implementiert werden müssen. Die Operatoren für die Werte von Integer-Objekten innerhalb von `NumericFDDomainElement` werden von Java zur Verfügung gestellt. Für Berechnungen und Vergleiche mit `RealInterval`-Objekten innerhalb von `IntervalDomainElement` wird die in der Bibliothek IAMath implementierte Intervallarithmetik eingesetzt. Während Berechnungen und Vergleiche für numerische finite und Intervalldomänen daher unproblematisch sind, liefern für symbolische finite Domänen bisher ausschließlich der Gleichheits- und Ungleichheitsoperator innerhalb von `SymbolicFDDomainElement` sinnvolle Ergebnisse in Form von String-Vergleichen.

Da sich die Instanzen der Unterklassen von `DomainElement` innerhalb von `Domain` bzw. den Unterklassen von `Domain` befinden, benötigt `DomainElement` ebenfalls eine Methode `cloneDeep()`, um ggf. eine vollständige Duplizierung des enthaltenden, übergeordneten Objekts sicherstellen zu können (vgl. Abschnitt 7.5.1, S. 205 und Abschnitt 7.5.2, S. 207).

7.6 Implementierte Constraint-Lösungsverfahren

In der Abbildung 7.6 auf Seite 211 ist eine Übersicht über die Klassenhierarchie des Unterpackages `yacs.solver` und der implementierten Constraint-Lösungsverfahren zu sehen. Sämtliche Constraint-Solver müssen das Interface `Solver` implementieren, in der die Methode `evaluate()` definiert wird. Je nachdem, ob ein Solver für das Preprozessing, für die Konsistenzherstellung oder die Lösungssuche eingesetzt werden soll, muss die entsprechende, abstrakte Oberklasse spezialisiert werden: `PreprocessingSolver`, `ConsistencySolver` oder `SearchSolver`. Während die `evaluate()`-Methode von den abstrakten Solver-Oberklassen implementiert wird, deklariert jede Oberklasse, wiederum in Abhängigkeit von der entsprechenden Lösungsphase, eine abstrakte Methode, die von `evaluate()` aufgerufen wird und jeweils von den konkreten Constraint-Solvern implementiert werden muss: `process()`, `propagate()` und `search()`. Die derart gekapselten Lösungsalgorithmen realisieren ein Entwurfsmuster, das „Strategie" (engl. *strategy*) genannt wird (vgl. Gamma et al. 1996, S. 373 ff.). Auf diese Weise wird sichergestellt, dass der Constraint-Lösungsvorgang unabhängig von den tatsächlich zum Einsatz kommenden Lösungsalgorithmen durchgeführt werden kann. Die abstrakten Solver-Klassen bzw. das Solver-Interface sind Teil des Framework-Layers von YACS, die konkreten Implementierungen der Algorithmen sind entsprechend dem Algorithmus-Layer zugehörig (vgl. Abschnitt 6.5, S. 194).

Im Fokus der Umsetzung dieser Arbeit steht weniger die Implementierung von möglichst effizienten Verfahren zum Lösen von Constraint-Problemen, als vielmehr die flexiblen und dynamischen Aspekte innerhalb eines modularen Constraint-Frameworks. Entsprechend wurden an dieser Stelle lediglich eine Reihe einfacher Constraint-Lösungsverfahren realisiert, an denen die Funktionalität von YACS demonstriert werden kann. So wurde für Constraints mit finiten Domänen neben einfacher Knotenkonsistenz in der Klasse `NCSolver` bisher lediglich Kantenkonsistenz realisiert: `ACSolver` und `AC3Solver` (die letztere Klasse implementiert den `AC-3`-Algorithmus aus Abbildung 5.6, S. 95). Als Suchverfahren für FD-Constraints wurden zwei Backtracking-Suchalgorithmen implementiert: `SingleSolutionBTSolver` und `BacktrackingSolver` (für letztere Klasse siehe den Algorithmus in Abbildung 5.15, S. 117). Der `BacktrackingSolver` wurde mit vorhandenen Algorithmen für Knoten- und Kantenkonsistenz zu zwei MAC-Varianten um *look-ahead*-Mechanismen erweitert: `MACSolver` und `MAC3Solver`. Sowohl `BacktrackSolver` als auch die darauf basierenden `MACSolver` und `MAC3Solver` generieren über eine Hilfsklasse zu Beginn des Suchvorgangs eine statische Variablenordnung nach der `dom/deg`-Heuristik (vgl. Abschnitt 5.2.5, S. 128). Diese Heuristik ist eine Kombination des *fail-first*-Prinzips mit der *maximum-degree-ordering*-Heuristik, welche sich i. A. als recht effizient erwiesen hat (vgl. Bessière und Régin 1996, S. 69 ff.).

Spezielle Algorithmen zur Behandlung von n-ären FD-Constraints wurden bisher nicht implementiert, allerdings sind beide Backtracking-Solver in der Lage, n-stellige Constraint-Netze nach Lösungen zu durchsuchen. Die Algorithmen in `NCSolver` und `ACSolver` sind zudem „optimistisch“ implementiert: Constraints mit mehr als einer Variable (`NCSolver`) bzw. mehr als zwei Variablen (`ACSolver`) werden lediglich ignoriert, die anderen entsprechend auf Konsistenz überprüft und deren Domänen ggf. eingeschränkt. Die Implementierung in `AC3Solver` hingegen generiert eine *Exception*, falls das Constraint-Problem neben unären und binären auch höher-stellige Constraints aufweist. Analog gilt dies für `MACSolver` und `MAC3Solver`.

Für Intervalldomänen existiert in YACS mit der Klasse `HullConsistencySolver` ein Constraint-Solver, durch dessen Algorithmus Hull- bzw. 2B-Konsistenz für ein Constraint-Problem mit reellwertigen Intervalldomänen hergestellt werden kann (siehe Algorithmus in Abbildung 5.23, S. 152). Die Stelligkeit der Constraints ist hierbei irrelevant, allerdings ist der Solver derzeit noch recht einfach gehalten: Eine Zerlegung von Constraints wurde nicht implementiert, so dass für die korrekte Ausführung des Solvers als Eingabe ausschließlich vorbereitete *solution functions*, d. h. die impliziten Funktionen eines Constraints, zulässig sind (vgl. Abschnitt 5.3.4.1, S. 155). Außerdem lassen sich durch die Intervallverarbeitung von YACS zurzeit ausschließlich Wertebereiche von Intervallvariablen in *Gleichungen* einschränken. Constraint-Ausdrücke mit Ungleichungen werden implementierungsbedingt bisher nicht unterstützt.

Auf die Umsetzung von Preprozessing-Verfahren, wie bspw. die Transformation n-ärer in binäre (FD-)Constraints durch Einführung „umfassender Variablen“ (vgl. Abschnitt 5.2.6.2, S. 138) oder die Zerlegung von komplexen (reellwertigen) Constraint-Aus-

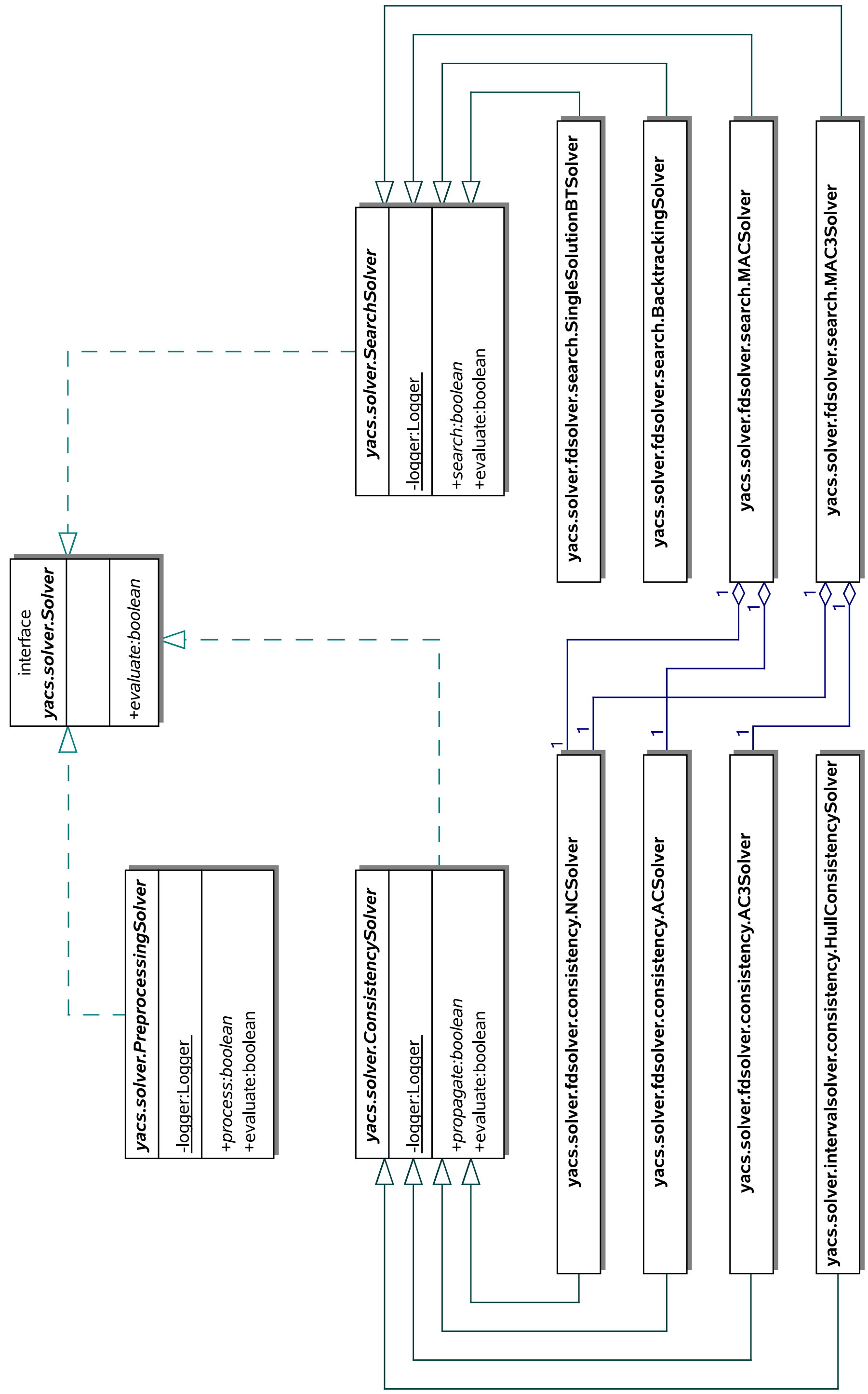


Abbildung 7.6: Übersicht über die implementierten Constraint-Lösungsverfahren

drücken als Vorstufe zur Herstellung von *solution functions* (vgl. Abschnitt 5.3.4.1, S. 155) bzw. Projektionen (vgl. Definition 5.3.14, S. 160), wurde verzichtet. Eine Implementierung derartiger Mechanismen würde über den Rahmen dieser Arbeit hinausgehen und ist nicht notwendig, um die Funktionalität des YACS-Frameworks zu demonstrieren.

Erwähnenswert ist, dass die FD-Solver von YACS in der Lage sind, sowohl numerische als auch symbolische Wertebereiche zu verarbeiten. Durch die Kapselung der Elemente und deren Operatoren in den Wertebereichen, spielt es keine Rolle, welche Art Werte propagiert werden (vgl. Abschnitt 7.5.3, S. 208). Außerdem ist zu beachten, dass aufgrund des globalen Namensraums Variablen mit demselben Namen, die von Constraint-Solvern in unterschiedlichen Strategien verarbeitet werden, YACS-intern dasselbe Objekt darstellen. Dies kann unbeabsichtigte Seiteneffekte hervorrufen, daher sollte bei der Namensgebung von Variablen dieser Umstand entsprechende Beachtung erfahren.[11]

Die im YACS-Framework bisher implementierten Constraint-Lösungsverfahren werden an dieser Stelle übersichtlich mit einer kurzen Beschreibung aufgeführt:

- `NCSolver`: Ein Constraint-Solver zum Herstellen von Knotenkonsistenz über finite Domänen nach Marriott und Stuckey (1999, S. 94).

- `ACSolver`: Ein Constraint-Solver zum Herstellen von Kantenkonsistenz über finite Domänen nach Marriott und Stuckey (1999, S. 94).

- `AC3Solver`: Ein Constraint-Solver zum Herstellen von Kantenkonsistenz über finite Domänen mittels `AC-3` nach Mackworth (1977a, S. 106).

- `HullConsistencySolver`: Beinhaltet einen Algorithmus zum Herstellen von Hull-Konsistenz bzw. 2B-Konsistenz für Variablen mit Wertebereichen aus reellwertigen Intervallen. Der Algorithmus orientiert sich am Waltz-Filteralgorithmus und den Beschreibungen von Davis (1987), Hyvönen (1992) und Lhomme (1993). Eine Zerlegung der Constraints findet nicht statt, außerdem lassen sich ausschließlich Constraints verarbeiten, die bereits vollständig als implizite Funktionen (*solution functions*, vgl. Abschnitt 5.3.4.1, S. 155) bzw. als „Projektionen" vorliegen (vgl. Definition 5.3.14, S. 160). Implementierungsbedingt lassen sich derzeit ausschließlich implizite Funktionen verarbeiten, die als *Gleichungen* vorliegen.

- `SingleSolutionBTSolver`: Ein rekursiver Backtracking-Solver für finite Domänen in Anlehnung an Marriott und Stuckey (1999, S. 90). In dieser Implementierung allerdings wird nicht nur `true/false` zurückgegeben, sondern ggf. eine Lösung generiert. Dies ist die erste Lösung, die durch den Suchalgorithmus aufgefunden wird. Der Solver eignet sich in erster Linie für die Überprüfung, ob ein Problem inkonsistent ist oder nicht.

[11] Für die Behandlung von umfangreichen Problemstellungen, für die eine separate Verarbeitung in voneinander getrennten Constraint-Netzen sichergestellt werden soll, würde sich die Verwendung von Präfixen für die Namen von Constraint-Variablen anbieten. Ein Präfix könnte bspw. der Name oder ein Kürzel für die jeweilige Constraint-Lösungsstrategie sein.

- `BacktrackingSolver`: Ein Backtracking-Solver zur Lösungssuche in einem Constraint-Problem mit finiten Domänen nach Dechter und Frost (1998, S. 13) bzw. Dechter und Frost (2002, S. 152). Der Solver wurde dahingehend modifiziert, dass anstatt lediglich einer Lösung alle möglichen Lösungen eines Problems gefunden werden.

- `MACSolver`: Ein MAC-Solver zur Lösungssuche mit *look-ahead* in einem Constraint-Problem mit finiten Domänen nach Dechter und Frost (2002, S. 175 u. 178). Der Solver ist eine Erweiterung des `BacktrackingSolver`. Als *look-ahead*-Mechanismus wird `ACSolver` eingesetzt. Ebenso wie `BacktrackingSolver` ist `MACSolver` in der Lage, anstatt lediglich einer Lösung alle Lösungen eines Problems zu generieren.

- `MAC3Solver`: Ein MAC-3-Solver zur Lösungssuche mit *look-ahead* in einem Constraint-Problem mit finiten Domänen nach Dechter und Frost (2002, S. 175 u. 178). Der Solver ist ebenfalls eine Erweiterung von `BacktrackingSolver`. In diesem Fall kommt `AC3Solver` als *look-ahead*-Mechanismus zum Einsatz. Auch `MAC3Solver` ist als vollständiger Solver implementiert, d. h. er ist ebenso wie `BacktrackingSolver` in der Lage, sämtliche Lösungen eines Problems zu finden.

Wie aus dem UML-Diagramm in Abbildung 7.6 auf Seite 211 hervorgeht, nutzen die beiden MAC-Suchalgorithmen die vorhandenen Konsistenz-Solver für das erforderliche *look-ahead* direkt, indem sie die benötigten Klassen aggregieren. Zur Laufzeit werden den instantiierten Konsistenz-Solvern in der *look-ahead*-Phase die jeweils einzuschränkenden Constraints übergeben und die Propagierung gestartet. Je nach Problemstellung und Härte des Problems, kann der Aufwand für das durchgeführte *look-ahead* potentiell unnötiger Overhead sein (vgl. Abschnitt 4.3, S. 54 und Abschnitt 5.2.4.6, S. 122). Generell ist für Probleme mit vielen Lösungen und wenig Filtermöglichkeiten das einfache Backtracking den MAC-Algorithmen vorzuziehen.

7.7 Constraint-Lösungsstrategien

Elementare Bedeutung innerhalb des YACS-Frameworks kommt den Constraint-Lösungsstrategien zu. In ihnen wird definiert, welche Lösungsverfahren in welcher Reihenfolge anzuwenden sind (vgl. Abschnitt 6.3, S. 179). Die nachfolgend aufgeführten Klassen sind Teil des Strategie-Layers von YACS (vgl. Abschnitt 6.5, S. 194).

Die Constraint-Lösungsstrategien werden für YACS innerhalb einer XML-Datei abgelegt. Der Name dieser Datei ist frei wählbar. Wird bei der Instantiierung des YCM kein Dateipfad angegeben, so wird standardmäßig die Datei im aktuellen Verzeichnis unter dem Namen `yacs_strategies.xml` gesucht.

7.7.1 Der XML-Parser

Der Parser zum Einlesen der Lösungsstrategien ist in der Klasse `StrategyReader` implementiert (siehe Abbildung 7.7 auf der nächsten Seite). Der Konstruktor der Klasse `StrategyReader` bekommt vom YCM den Dateipfad zur XML-Datei übergeben und liest

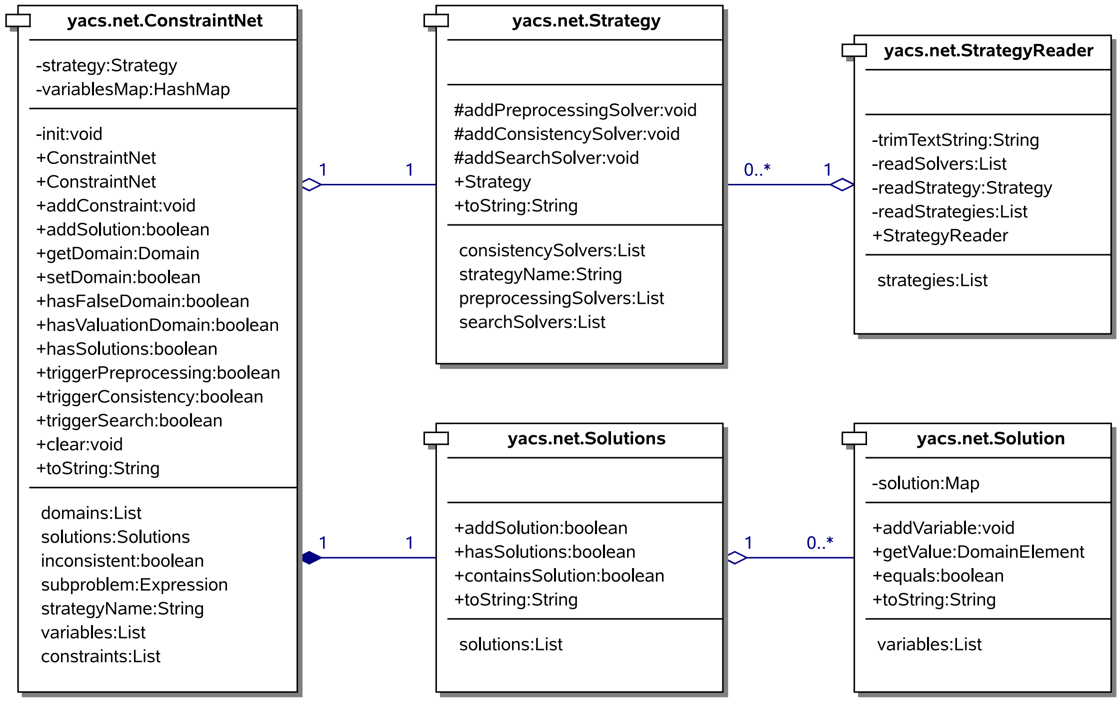


Abbildung 7.7: Constraint-Lösungsstrategien und Constraint-Netze

die darin gespeicherten Informationen aus. Der XML-Parser ist ein sog. *Validating Parser*, d. h. die Konsistenz der XML-Daten wird anhand einer *Document Type Definition* (DTD) überprüft.

In Anhang C auf Seite 278 ff. ist die verwendete DTD sowie eine Beispieldatei mit Definitionen für einfache Strategien dokumentiert. Entsprechend der DTD enthalten Strategien neben ihrem Namen als Parameter drei Abschnitte, welche die jeweilige Phase des Constraint-Lösungsprozesses repräsentieren: `preprocessing`, `consistency` und `search`. In jedem dieser Abschnitte lassen sich eine Reihe von Constraint-Solvern anhand ihres Klassennamens (inkl. des vollständigen Package-Pfades) spezifizieren.[12] Diese Solver-Klassen werden vom `StrategyReader` instantiiert und einer Instanz der Klasse `Strategy` übergeben (siehe Abbildung 7.7). Jede Instanz von `Strategy` repräsentiert anschließend eine Strategie, wie sie innerhalb der XML-Datei spezifiziert wurde.

7.7.2 Einbettung in Constraint-Netze

In der Implementierung von YACS wird jedem primitiven Constraint-Ausdruck eine Lösungsstrategie zugewiesen, die für die Auswertung des jeweiligen Constraints zuständig ist. Dies führt zur Bildung von Teilproblemen bzw. Constraint-(Teil-)Netzen, die durch

[12]Die spezifizierten Constraint-Solver müssen die jeweilige abstrakte Solver-Klasse implementieren (vgl. Abschnitt 7.6, S. 209).

die Klasse `ConstraintNet` repräsentiert werden (siehe Abbildung 7.7 auf der vorherigen Seite).

Die Instanzen der Klasse `ConstraintNet` werden vom YCM erzeugt und verwaltet. Für jede vom `StrategyReader` gelieferte Strategie wird ein Constraint-Netz generiert und mit der entsprechenden Strategie belegt. Jedem Teilproblem bzw. jedem Constraint-Netz ist anschließend anhand der jeweils vorhandenen Strategie bekannt, welche Constraint-Solver in welcher Phase einzusetzen sind.

Um Ergebnisse von Suchalgorithmen speichern zu können, ist jedem Constraint-Netz eine Instanz der Klasse `Solutions` zugeordnet. Innerhalb dieser Container-Klasse können durch Constraint-Solver Lösungsobjekte, namentlich Instanzen der Klasse `Solution`, abgelegt werden (siehe Abbildung 7.7 auf der vorherigen Seite).

7.7.3 Ausführungskontrolle

Zur Auswertung der Constraints, d. h. zur Anwendung der Constraint-Lösungsstrategien, dient die Methode `evaluate()` des YCM (siehe Abbildung 7.2, S. 204). Der Constraint-Manager verwaltet intern eine Liste mit den existierenden Constraint-Netzen, welche sich über den Namen der jeweils zugehörigen Strategie referenzieren lassen. Zur Anwendung der Constraint-Lösungsstrategien werden innerhalb von `evaluate()` für jede Phase des Lösungsprozesses sämtliche Constraint-Netze der Reihe nach durchgegangen, und die der Phase entsprechende Methode des Constraint-Netzes zur Anwendung der spezifizierten Constraint-Solver aufgerufen: Die Anwendung der Preprozessing-Solver geschieht durch den Aufruf der Methode `triggerPreprocessing()` der Klasse `ConstraintNet` (siehe Abbildung 7.7 auf der vorherigen Seite). Konsistenz wird durch den Aufruf der Methode `triggerConsistency()` in derselben Klasse hergestellt. Die Lösungssuche wird entsprechend mit `triggerSearch()` gestartet.

Die Methode `triggerSearch()` unterscheidet sich von `triggerPreprocessing()` und `triggerConsistency()` dadurch, dass zu Beginn von `triggerSearch()` ggf. existierende Lösungen der Constraint-Netze entfernt werden. Da zwischen mehreren Aufrufen der Methode `evaluate()` sowohl die Constraint-Netze als auch die Wertebereiche von Constraint-Variablen modifiziert werden können, wird auf diese Weise sichergestellt, dass keine potentiell inkonsistenten Lösungen für ein Teilproblem vorgehalten werden.

Der Ablauf der Methode `evaluate()` des Constraint-Managers ist in dem Aktivitätsdiagramm in Abbildung 7.8 auf der nächsten Seite dargestellt. Der Aufruf der jeweiligen *trigger*-Methoden erfolgt in jeder Phase solange, bis von den entsprechenden Constraint-Solvern keine Einschränkungen der Wertebereiche mehr gemeldet werden, dargestellt durch das Flag `domainModification`. Die wiederholten Iterationen sind notwendig, damit Wertebereichseinschränkungen innerhalb eines Constraint-Netzes von den übrigen Constraint-Netzen berücksichtigt werden können.[13] Durch diese „Metapropagation“ profitieren die vorhandenen Constraint-Netze von den Wertebereichseinschränkungen untereinander und erreichen insgesamt eine erhöhte Filterrate inkonsistenter Werte.

[13]Aufgrund des vorhandenen globalen Namensraums für Constraint-Variablen (vgl. Abschnitt 7.5, S. 204).

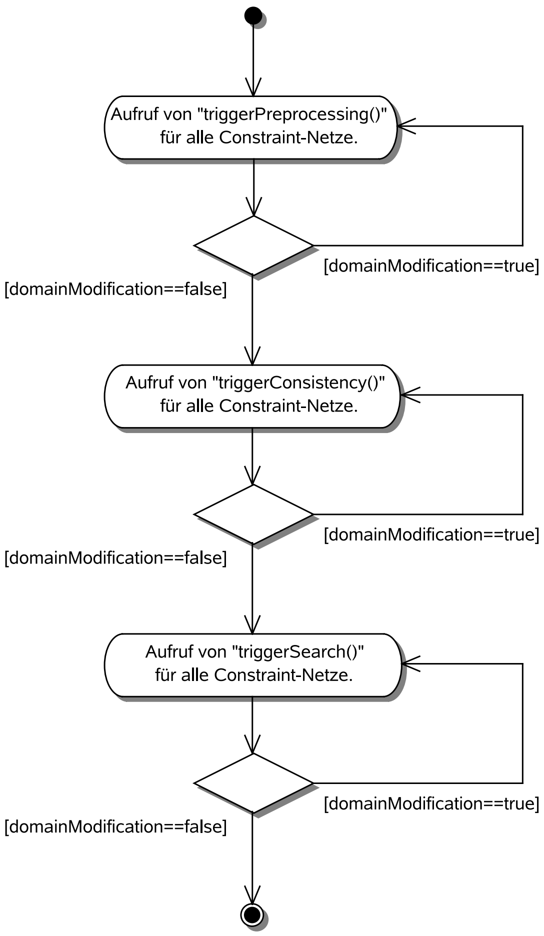


Abbildung 7.8: Schematische Darstellung des Constraint-Lösungsvorgangs von YACS

An dieser Stelle wird deutlich, dass die Unterscheidung des Lösungsprozesses in unterschiedliche Phasen semantisch einen Unterschied macht, wenn auch syntaktisch keine Unterscheidung feststellbar ist (die Schnittstellen für die Constraint-Solver sind identisch). Bei Überlappungen von Constraint-Netzen haben Wertebereichseinschränkungen Einfluss auf benachbarte Constraint-Netze, allerdings ausschließlich innerhalb einer Phase des Lösungsprozesses. Es macht daher Sinn, z. B. einen Algorithmus zur Konsistenzherstellung auch wirklich als „ausgewiesenen“ Konsistenz-Solver innerhalb von YACS zu nutzen (durch Spezialisierung der entsprechenden abstrakten Oberklasse und dem entsprechenden Eintrag innerhalb einer Constraint-Lösungsstrategie), da er ansonsten nicht von den Wertebereichseinschränkungen anderer Konsistenz-Solver profitiert und umgekehrt.

Nicht unbedingt offensichtlich ist, warum auch die Constraint-Solver während der Phase der Lösungssuche eine Änderung der Wertebereiche anzeigen können. In der Tat liefern alle derzeit implementierten Suchalgorithmen an dieser Stelle ausschließlich `false` zurück. Zum einen besteht aus Gründen einer einheitlichen Schnittstelle auch hier die Möglichkeit der Änderungsanzeige, zum anderen wird dadurch die Möglichkeit gegeben, dass Suchalgorithmen ggf. Wertebereichseinschränkungen vornehmen (z. B. durch eine Algorithmus-interne, vorgeschaltete Konsistenzherstellung). Andere Suchalgorithmen können auf diesem Weg davon Kenntnis erlangen, dass es (weitere) inkonsistente Werte gibt, die nicht Teil einer (globalen) Lösung sein können.[14]

Um auch phasenübergreifend Wertebereichseinschränkungen propagieren zu können, wäre eine weitere, äußere Iteration des Kontrollalgorithmus aus Abbildung 7.8 auf der vorherigen Seite notwendig. In Bezug auf eine globale, Constraint-Netz übergreifende Lösungssuche wäre es denkbar, dass Suchalgorithmen, die für einzelne Constraint-Netze (Teilprobleme) lokal konsistente Lösungen generiert haben, aus diesen Lösungen konsistente Wertebereiche für die involvierten Constraint-Variablen ableiten. In Bezug auf das jeweilige Teilproblem sind diese Wertebereiche „global“ konsistent (vollständige Suchverfahren vorausgesetzt), und würden bei einer erneuten, äußeren Iteration eines erweiterten Kontrollalgorithmus ggf. weitere Wertebereichseinschränkungen durch die jeweilige Metapropagation der anderen Phasen erreichen.[15]

So gesehen ist die derzeitige Implementierung des Kontrollalgorithmus des YACS-Frameworks eine Vorstufe für die Lösungssuche im globalen Kontext. Wohlgemerkt betrifft dies ausschließlich die globale *Propagation*, denn nach dem oben beschriebenen Schema lassen sich immer noch ausschließlich lokale Lösungen generieren (auch wenn bereits global propagiert wird). Eine globale, Constraint-Netz übergreifende Lösungssuche macht eine übergeordnete Suche innerhalb eines Meta-Constraint-Solvers notwendig, wie er in Abschnitt 6.4.3 auf Seite 191 ff. beschrieben wurde. Einen solchen bietet die derzeitige Realisierung von YACS nicht. Aus Effizienzgründen wurde daher auf eine weitere, äußere Iteration bisher verzichtet, und der Kontrollalgorithmus stattdessen in seiner jetzigen Form belassen.

7.7.4 Beispiel für einen Constraint-Lösungsvorgang

Ein einfaches Beispiel für einen Lösungsvorgang in der derzeitigen Implementierung des YCM ist in dem Sequenzdiagramm in Abbildung 7.9 auf der nächsten Seite zu sehen:

- Die `evaluate()`-Methode des Constraint-Managers wird in Schritt 1 von außerhalb aufgerufen.

- In Schritt 2 und 3 wird jeweils das Preprozessing der beiden vorhandenen Constraint-Netze mit den zugeordneten Strategien namens `medium_consistency` und `search` aufgerufen. Durch beide Aufrufe erfolgt keinerlei Änderung an den Domänen der

[14] In diesem Fall kann allerdings für ein Teilproblem, für das in einem vorherigen Schritt einer Lösungsstrategie ein bestimmter Konsistenzgrad hergestellt wurde, und das Überlappungen mit einem Teilproblem aufweist, auf welches eine derartige Lösungssuche angewendet wurde, nach Anwendung der Suche dieser Konsistenzgrad nicht mehr garantiert werden.

[15] Zum Beispiel durch eine Strategie, die keinen Constraint-Solver zur Lösungssuche enthält, allerdings Überlappungen mit Variablen eines Constraint-Netzes aufweist, deren Strategie eine Suche vorsieht. So wie die Variablen des Constraint-Netzes dieser Strategie ggf. weitere Wertebereichseinschränkungen erfahren, haben hier definierte Constraints ggf. Auswirkungen auf Constraint-Netze mit (globaler) Lösungssuche.

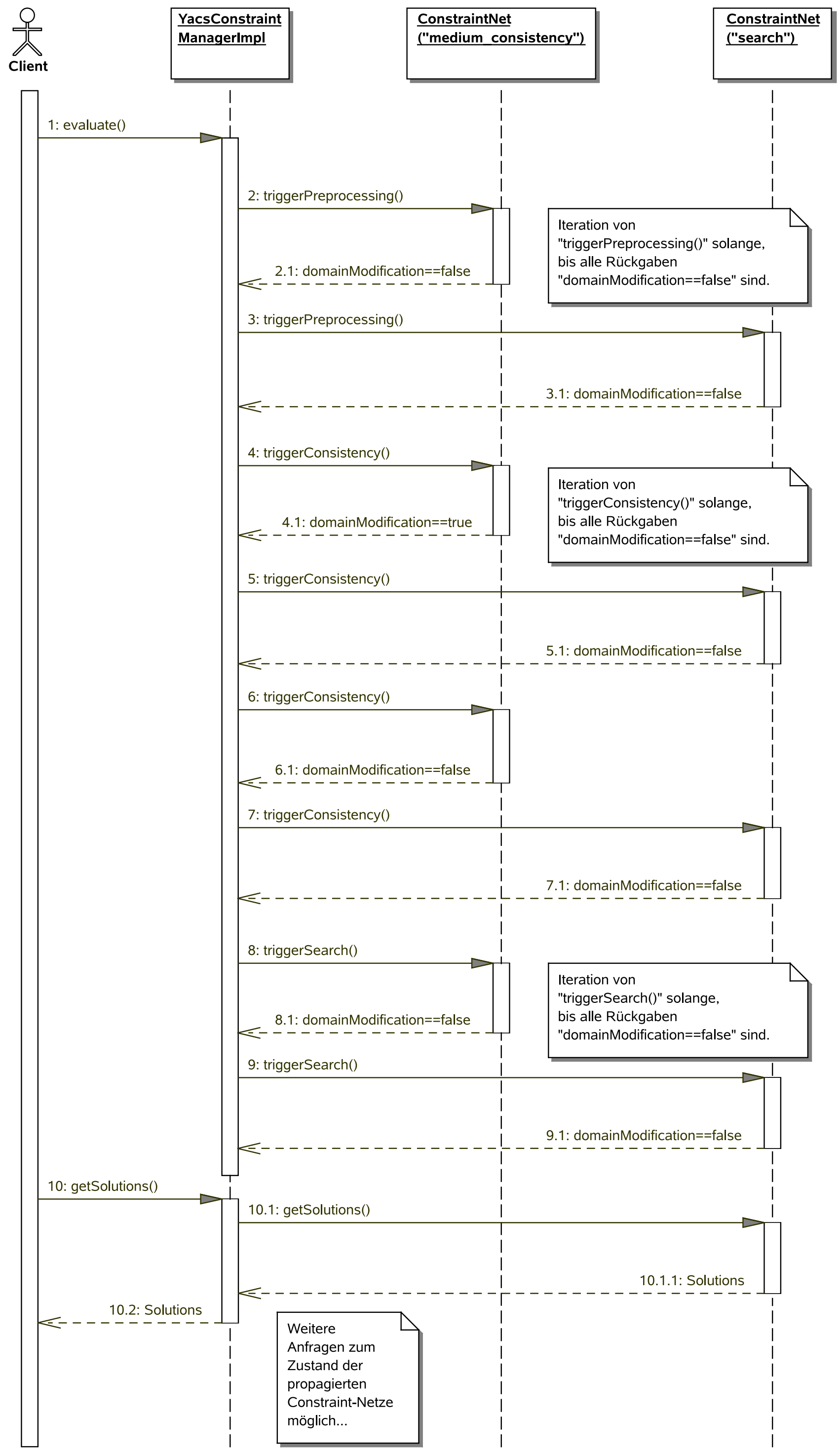


Abbildung 7.9: Beispiel für einen Constraint-Lösungsvorgang

beteiligten Constraint-Variablen, daher kann in Schritt 4 und 5 mit der Konsistenzherstellung fortgefahren werden.

- Als Ergebnis von Schritt 4 liefert die Propagation des Constraint-Netzes, dem die Strategie `medium_consistency` zugeordnet ist, eine Einschränkung der Wertebereiche (`domainModification==true`). Daher ist eine erneute Iteration zur Konsistenzherstellung notwendig (Schritt 6 und 7).

- Anschließend erfolgt die Lösungssuche, welche ebenfalls keine Änderung der Wertebereiche zur Folge hat. Die Constraint-Auswertung ist damit beendet.

Es können abschließend Anfragen an den Constraint-Manager gestellt werden, die Informationen über gefundene Lösungen zur Antwort haben. Außerdem lassen sich z. B. die Wertebereiche bestimmter Variablen oder der Zustand eines bestimmten Constraint-Netzes abfragen. Es ist weiterhin möglich, den Constraint-Netzen weitere Constraints hinzuzufügen oder die Wertebereiche existierender Constraint-Variablen manuell zu modifizieren und einen erneuten Auswertungsvorgang zu starten.

7.8 Ausnahmebehandlung

Für die Ausnahmebehandlung stehen innerhalb von YACS eine Reihe von *Exceptions* zur Verfügung. Beispielsweise erzeugen die implementierten Constraint-Solver eine `InconsistencyException`, wenn das zu verarbeitende Teilproblem inkonsistent ist. In dem entsprechenden Constraint-Netz wird anschließend das Flag `inconsistent` mit `true` belegt, welches über die Methode `isInconsistent()` der Klasse `ConstraintNet` abgefragt werden kann (vgl. Abbildung 7.7, S. 214). Der XML-Parser für die Constraint-Lösungsstrategien generiert im Fehlerfall eine `StrategyParserException`. Für den Constraint-Parser ist für Ausnahmesituationen die Exception mit dem Namen `ConstraintParserException` vorgesehen.

Die für den Benutzer relevanten Exceptions werden an der Schnittstelle zum YCM entsprechend durchgereicht. Es handelt sich hierbei i. d. R. um eine `StrategyNotFoundException` oder eine `VariableNotFoundException`, die jeweils in Reaktion auf eine dem YACS-Framework unbekannte Strategie bzw. Variable ausgelöst werden. Für eine ausführlichere Beschreibung der vorgesehenen bzw. zur Verfügung stehenden Exceptions sei an dieser Stelle auf den Abschnitt E.9 auf Seite 313 der API-Dokumentation verwiesen.

7.9 Integration

Das im Rahmen dieser Arbeit erstellte Constraint-Framework YACS wurde im Austausch für den bestehenden Constraint-Solver prototypisch an das Konfigurierungswerkzeug EngCon angebunden. Außerdem ist die Schnittstelle von YACS allgemein gehalten, so dass das Framework auch in andere Anwendungen, die ein Constraint-System als Inferenzmechanismus benötigen, möglichst problemlos integrierbar ist.

7.9.1 Anbindung an EngCon

Die Anbindung an EngCon gestaltet sich derart, dass ein Wrapper erstellt werden muss, der eine Umsetzung der von EngCon geforderten Methoden auf die Schnittstelle von YACS übernimmt. Eine solche Wrapper-Klasse existiert bereits in EngCon für den derzeit angebundenen, externen Constraint-Solver. Die Schnittstelle zum Constraint-System von EngCon sieht vor, dass das Interface `ConstraintSolverRmiInterface` im Package `engcon.control.constraintControl` implementiert werden muss, um die Anbindung eines Constraint-Solvers zu gewährleisten. In der derzeitigen Anbindung wird dies mit der Implementierung in der Klasse `ConstraintSolverRmiInterfaceImpl`, welche sich im selben Package wie das obige Interface befindet, durch Anbindung einer externen C++-Bibliothek per *Java Native Interface* (JNI)[16] erreicht (vgl. Stearns 1997, 1998).

Die obige Klasse wurde für die Integration von YACS in EngCon durch eine eigene Implementierung ausgetauscht: Die Klasse `YacsEngConInterface` implementiert ebenfalls das Interface `ConstraintSolverRmiInterface`, nutzt dabei allerdings die Constraint-Lösungsmöglichkeiten des YACS-Frameworks. Die Implementierung dieser Schnittstelle ist prototypisch. Es wurden die essentiellen Methoden implementiert, die übrigen Methoden sind z. T. „Dummys". Diese Anbindung von YACS an EngCon dient damit ausschließlich Demonstrationszwecken und müsste für einen tatsächlichen Einsatz vervollständigt werden.

Die Implementierung in `YacsEngConInterface` sieht vor, dass die vorhandenen Wissensbasen von EngCon für eine Zusammenarbeit mit YACS geringfügig modifiziert werden müssen: Jeder Constraint-Ausdruck eines Funktions-Constraints muss, separiert durch ein Trennzeichen („#"), um den Namen der jeweils anzuwendenden Constraint-Lösungsstrategie erweitert werden. Auf diese Weise lassen sich flexibel für unterschiedliche Constraints einer Wissensbasis unterschiedliche Lösungsverfahren einsetzen. Eine Alternative bestünde darin, eine global einzusetzende Lösungsstrategie statisch innerhalb der Wrapper-Klasse zu implementieren. Bei einer derartigen, statischen Lösung könnte auf die Modifikation der Wissensbasen verzichtet werden. Zugunsten der Flexibilität, und insbesondere um Lösungsverfahren sowohl für finite als auch infinite Wertedomänen anwenden zu können, wurde der ersteren Lösung der Vorzug gegeben.

Als problematisch erwies sich, dass EngCon Funktions-Constraints für finite und infinite Domänen bisher nicht unterscheidet. So existiert innerhalb von EngCon lediglich eine einzige Container-Klasse zur Kapselung und Übertragung der Wertebereiche von Constraint-Variablen an den Constraint-Solver. Auch diskrete Werte werden innerhalb dieser Klasse als Intervalle repräsentiert, nämlich als diskontinuierliche Punktintervalle.[17] Dies war in der Vergangenheit unproblematisch, da der bisher in EngCon integrierte Constraint-Solver ausschließlich reellwertige Intervalle verarbeitet. Innerhalb von YACS existieren allerdings unterschiedliche Kapselungen für unterschiedliche Wertedomänen. Zum einen ist eine Vermischung in der derzeitigen Implementierung von YACS nicht vorge-

[16] `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`

[17] Symbolische Domänen für Funktions-Constraints finden innerhalb von EngCon bisher keinerlei Berücksichtigung.

sehen, zum anderen sollten die jeweiligen Domänen schon aus Effizienzgründen in ihrer entsprechenden Repräsentationsform genutzt werden.

Vor jeder Propagation der Constraint-Netze werden die Wertebereiche der Constraint-Variablen an YACS übergeben. Zur Vermeidung von Ausnahmen in Form von `ClassCast-Exceptions` in YACS, wurde die Unterscheidung von Domänen in der prototypischen Anbindung an EngCon innerhalb von `YacsEngConInterface` pragmatisch umgesetzt: Wenn das übergebene Container-Objekt eines Wertebereichs mehrere Elemente enthält oder bereits eine FD-Variable mit gleichem Namen existiert, wird eine numerische FD-Domäne erzeugt. Ist im Container-Objekt lediglich ein Element enthalten, wird davon ausgegangen, dass es sich um ein konvexes Intervall handelt und entsprechend eine Intervalldomäne erstellt.[18]

Diese einfache Implementierung einer Integration von YACS in EngCon ist ausreichend, um die im folgenden Kapitel 8 vorgestellten Constraint-Probleme zu verarbeiten. Ein stabiler Betrieb im produktiven Einsatz kann mit dieser Integration allerdings nicht sichergestellt werden. Für eine weitergehende Unterscheidung der Wertebereiche müsste die interne Repräsentation von EngCon entsprechend erweitert werden. Da dies über die Implementierung einer Schnittstelle hinausgeht und es sich zudem um den Forschungsprototyp EngCon V0 handelt, wurde von einer entsprechenden Erweiterung an dieser Stelle abgesehen.

7.9.2 Integration in andere Systeme

Wie für EngCon müssten für andere Anwendungen ebenfalls Wrapper-Klassen implementiert werden, wenn das YACS-Framework eingebunden werden soll. Die Schnittstelle von YACS ist hierfür möglichst schlank und „generisch" gehalten, so dass sich das Framework auch in andere Anwendungen, die einen stringbasierten Zugriff auf ein Constraint-System benötigen, problemlos integrieren lassen sollte.

Die für den reinen Anwender vorgesehene Schnittstelle von YACS beschränkt sich auf den Constraint-Manager. Diese einfache Schnittstelle kann als „äußere Schnittstelle" bezeichnet werden. Nutzer von YACS, die das Framework erweitern möchten, z. B. durch zusätzliche Constraint-Solver, sind auf die komplexere, „innere Schnittstelle" von YACS angewiesen. Diese umfasst neben der Klasse `ConstraintNet` die Klassen `Expression`, `Domain`, `DomainElement` und `Solver` respektive deren entsprechenden Unterklassen. Außerdem ist das Exception-Handling zu berücksichtigen.

Anwender, die das YACS-Framework ohne die integrierte, stringbasierte Schnittstelle nutzen möchten, müssen ebenfalls auf die internen Klassen von YACS zurückgreifen. Diese Art der Nutzung von YACS ist allerdings umständlicher, da Constraint-Ausdrücke, die ansonsten durch den Constraint-Parser generiert würden, manuell erstellt werden müssen.

Im dem folgenden Kapitel 8 wird u. a. anhand des Programms `YacsTester` (vgl. Anhang F, S. 315) dargelegt, wie sich die einfache, äußere Schnittstelle des YACS-Frameworks

[18] Der seltene Fall, dass zu Beginn, beim ersten Aufbau der Constraint-Netze, eine FD-Variable angelegt werden soll, die nur ein Element in ihrer Domäne aufweist, kann mit dieser simplen Unterscheidung nicht erkannt werden.

nutzen lässt, um Constraint-Probleme zu propagieren bzw. um Lösungen für die Problemstellungen zu generieren.

Kapitel 8

Validierung von YACS

You cannot prove a theory, but you can disprove it.

KARL R. POPPER

An dieser Stelle erfolgt eine Validierung der Umsetzung von YACS anhand der realisierten Funktionalität und der zuvor benannten Anforderungen. Neben synthetischen Problemstellungen wird die Integration des YACS-Frameworks in das strukturbasierte Konfigurierungswerkzeug ENGCON validiert. Außerdem erfolgt eine Positionierung im Vergleich mit weiterführenden Arbeiten.

8.1 Einleitung

Im Folgenden wird erläutert, welche Teile der Konzeption aus Kapitel 6 mit der Implementierung von YACS realisiert wurden. Daneben erfolgt eine Validierung sowohl anhand von „synthetischen“ Problemstellungen als auch im praktischen Einsatz mit dem strukturbasierten Konfigurierungswerkzeug ENGCON. Der Intention dieser Arbeit folgend liegt der Fokus dabei weniger auf der Leistungsfähigkeit der implementierten Constraint-Lösungsverfahren, als vielmehr darauf, ob die erstellte Lösung hinsichtlich der Flexibilität und Modularität den Anforderungen genügt.

Im Zuge der Erstellung dieser Arbeit gerieten zunehmend weitergehende Lösungen in den Blickwinkel. So lässt sich neben einer flexiblen Kombination von Constraint-Solvern ebenso die flexible Kooperation unterschiedlicher Lösungsmethoden betrachten. Dafür sprechen neben Effizienzgründen die Anwendbarkeit derartiger Systeme auf ein größeres Problemspektrum. Diesen Aspekten und einer Abgrenzung zu dem in dieser Arbeit entwickelten System ist daher ein eigener Abschnitt gewidmet.

8.2 Eigenschaften des Prototypen

YACS verfügt durch die strikte Ausnutzung objektorientierter Entwurfsmethoden über eine modulare Framework-Architektur. Es wurden Abstraktionen von konkreten Implementierungen gebildet, um den problemlosen Austausch bzw. die Erweiterung spezieller Komponenten zu ermöglichen. Konkrete Implementierungen hingegen wurden gekapselt, wodurch eine flexible Anbindung und die problemabhängige Austauschbarkeit gewährleistet wird. Über definierbare Constraint-Lösungsstrategien lassen sich unterschiedliche Constraint-Lösungsverfahren kombinieren und problemabhängig einsetzen. Das YACS-Framework unterstützt dabei sowohl finite als auch infinite Domänen sowie ein inkrementell anwachsendes Constraint-Netz.

Von den in Abschnitt 6.6 auf Seite 196 ff. genannten vier Szenarien für eine Umsetzung des vorgelegten Konzepts wurde das erste und zweite Szenario erreicht bzw. übertroffen. Die Voraussetzungen für die Szenarien drei und vier werden bisher lediglich zum Teil erfüllt. Es wurden allerdings bereits entsprechende Vorbereitungen getroffen, die für eine vollständige Realisierung notwendig sind.

In der Implementierung von YACS lassen sich beliebig viele Constraint-Lösungsstrategien definieren und einsetzen. Jeder primitive Constraint-Ausdruck kann theoretisch von einer eigenen Strategie verarbeitet werden. Überlappungen zwischen Constraints der selben Domäne sind möglich. Für deren Auswertung wird innerhalb von YACS eine Metapropagation realisiert. Diese ist auf die Nutzung eines einheitlichen Namensraums für sämtliche Constraint-Teilprobleme zurückzuführen. Durch wiederholtes Iterieren werden Wertebereichseinschränkungen in den beteiligten Constraint-Netzen propagiert (vgl. Abschnitt 7.7.3, S. 215). Die Metapropagation ist allerdings in der derzeitigen Implementierung von YACS nicht domänenübergreifend möglich. Es existieren zurzeit keine Konvertierungsroutinen, so dass eine Vermischung von Variablen mit unterschiedlichen Wertedomänen in denselben Constraints nicht möglich ist (vgl. Abschnitt 6.4.2, S. 186). Entsprechend wurden die Konzepte zum heterogenen Constraint-Lösen (vgl. Abschnitt 6.4.3, S. 191) bisher nicht realisiert.

Durch die Mittel zur Generierung eines globalen Namensraums wurden die Voraussetzungen zur Verarbeitung eines Meta CSPs geschaffen. Trotz der vorhandenen Metapropagation existiert bisher allerdings innerhalb des YACS-Frameworks kein eigenständiger Meta-Constraint-Solver für die Generierung von globalen Lösungen innerhalb eines Meta CSPs (vgl. Abschnitt 6.4.3, S. 191). Es ist daher möglich, dass Teillösungen von unterschiedlichen Constraint-Lösungsstrategien generiert werden, die sich gegenseitig ausschließen. Für den Anwendungsfall der strukturbasierten Konfigurierung mit EngCon ist dieser Umstand allerdings weniger relevant, da die Metasuche nach einer gültigen Konfiguration in diesem Fall durch die übergeordnete Suche während der Konfigurierung innerhalb von EngCon selbst vorgenommen wird. Entsprechend zählt die Verarbeitung eines Meta CSPs nicht zu den notwendigen Anforderungen für ein Constraint-System zur Unterstützung von strukturbasierten Konfigurierungsanwendungen.

Das mit dieser Arbeit erstellte Constraint-Framework wurde auf unterschiedliche Weise auf seine Funktionalität getestet. Neben „synthetischen“ Problemstellungen wurde es im Austausch für den bestehenden Constraint-Solver in einem Praxistest an das Konfigurierungswerkzeug EngCon angebunden. Auf die unterschiedlichen Testfälle wird im Folgenden eingegangen.

8.3 Validierung anhand synthetischer Probleme

Die Erstellung von synthetischen Problemen zum Testen von Constraint-Solvern ist ein übliches Vorgehen. Es sind eine Reihe von Problemgeneratoren verfügbar, wie z. B. der *Uniform Random Binary CSP Generator*[1] von *Christian Bessière*, mit denen sich Constraint-Probleme mit unterschiedlichen Eigenschaften komfortabel generieren lassen (vgl. Abschnitt 4.3, S. 54). Diesen Werkzeugen ist gemein, dass sich mit ihnen ausschließlich Constraint-Probleme in einer extensionalen Repräsentation generieren lassen. Dies ist problematisch, da das YACS-Framework ausdrücklich für intensionale Constraints entworfen wurde.

Einen Ausweg bietet eine Problembibliothek wie die CSPLib[2]. In ihr sind eine Reihe von Beispielproblemen aufgeführt, anhand derer sich die Effizienz von unterschiedlichen Constraint-Solvern vergleichen lässt. Im Fokus der Validierung des YACS-Frameworks steht allerdings weniger die Effizienz der implementierten Constraint-Solver. Vielmehr sollen grundlegendere Funktionalitäten des Prototypen von YACS getestet und demonstriert werden. Die synthetischen Probleme zur Validierung von YACS sind daher von relativ einfacher Natur und wurden den Lehrbüchern von Marriott und Stuckey (1999) sowie Russel und Norvig (2002) bzw. der Arbeit von Davis (1987) entnommen.

Im Rahmen der Entwicklung des YACS-Frameworks wurde das Programm `YacsTester` erstellt. Darin werden anhand einer Reihe von synthetischen Problemstellungen die Funktionalitäten des YACS-Frameworks und der entwickelten Lösungsverfahren demonstriert und getestet. Das Programm nimmt eine Initialisierung unterschiedlicher Constraint-Probleme vor und führt zwei Auswertevorgänge durch. Dazwischen werden Modifikationen an den bestehenden Constraint-Problemen vorgenommen. Die Ergebnisse der Auswertevorgänge werden jeweils ausgegeben.

Anhand des Programms `YacsTester` soll gezeigt werden, dass mit dem YACS-Framework unterschiedliche Problemstellungen strategiebasiert mit unterschiedlichen Lösungsverfahren behandelt werden können. Es soll weiterhin gezeigt werden, das sich durch YACS sowohl finite als auch infinite Wertebereiche verarbeiten lassen, wobei inkrementelle Veränderungen am Constraint-Netz möglich sein sollen.

Aus Gründen der Übersichtlichkeit ist das Programm `YacsTester` vollständig in Anhang F auf Seite 315 ff. dokumentiert. Die nachfolgenden Zeilenangaben beziehen sich auf das dort enthaltene Programm-Listing.

8.3.1 Übersicht über die Problemstellungen

Für das Programm `YacsTester` werden die Constraint-Lösungsstrategien aus Anhang C auf Seite 278 ff. genutzt. Für jede dort enthaltene Strategie existiert in `YacsTester`

[1] `http://www.lirmm.fr/~bessiere/generator.html`
[2] `http://www.csplib.org`

eine Methode, die ein Testproblem generiert und an den Constraint-Manager von YACS überträgt:

- In der Methode `NCSolverTest` (Zeile 114–160) wird die Strategie mit dem Namen `low_consistency` genutzt, um Knotenkonsistenz durch `NCSolver` herzustellen. Das in der Methode generierte Problem ist eine Erweiterung eines Beispielproblems von Marriott und Stuckey (1999, S. 95).
- Durch die Methode `AC3SolverTest` (Zeile 166–224) wird die Strategie `medium_consistency` eingesetzt, um durch die Kombination von `NCSolver` und `AC3Solver` für ein Kartenfärbeproblem einer Australienkarte strenge 2-Konsistenz herzustellen (vgl. Marriott und Stuckey 1999, S. 86 u. 89; Russel und Norvig 2002, S. 138).[3]
- `SingleSolutionBTSolverTest` (Zeile 230–280) nutzt die Strategie `simple_search` um eine einfache Backtracking-Suche auf ein Constraint-Beispiel von Marriott und Stuckey (1999, S. 89 u. 91) anzuwenden. Der in der Strategie spezifizierte `SingleSolutionBTSolver` gibt die erste Lösung zurück auf die er stößt, oder meldet entsprechend eine Inkonsistenz.
- Die `BacktrackingSolverTest` (Zeile 286–336) verwendet den in der Strategie `search` spezifizierten `BacktrackingSolver`, um sämtliche Lösungen für das *Smuggler's-Knapsack*-Problem zu generieren (vgl. Marriott und Stuckey 1999, S. 101).[4]
- Die in der Methode `MAC3SolverTest` (Zeile 342–392) genutzte Strategie mit dem Namen `search_with_look-ahead` versucht ebenfalls das *Smuggler's-Knapsack*-Problem zu lösen (vgl. Marriott und Stuckey 1999, S. 101), setzt dafür allerdings den `MAC3Solver` ein.
- In der Methode `HullConsistencySolverTest` (Zeile 398–439) wird der in der Strategie `interval_consistency` spezifizierte `HullConsistencySolver` eingesetzt, um das Constraint-Problem aus Beispiel 5.3.5 auf Seite 153 zu propagieren (vgl. Davis 1987, S. 286 f.). Weil `HullConsistencySolver` derzeit ausschließlich Gleichungen unterstützt, wurde das Problem entsprechend modifiziert. Außerdem müssen diesem Constraint-Solver die impliziten Funktionen eines Constraints (*solution functions*, vgl. Abschnitt 5.3.4.1, S. 155) vorgefertigt übergeben werden (Zeile 409–410).

Um Redundanzen zu vermeiden, wurden nicht alle implementierten Constraint-Solver in das Testprogramm `YacsTester` aufgenommen. Die Ergebnisse von `AC3Solver` und `MAC3Solver` unterscheiden sich nicht von den Ergebnissen von `ACSolver` und `MACSolver`, daher wurde auf die Darstellung letzterer an dieser Stelle verzichtet.

[3] Die Bundesstaaten einer Australienkarte (WA, NT, SA, Q, NSW, V, T) sind derart mit den Farben „rot“, „grün“ und „blau“ zu markieren, dass benachbarte Staaten in jedem Fall unterschiedliche Farben aufweisen.

[4] *Smuggler's Knapsack*: Ein Schmuggler hat einen Rucksack mit einer begrenzten Aufnahmefähigkeit von insgesamt 9 Wareneinheiten. Er kann während einer Schmuggeltour 4 Einheiten Whisky (W), 3 Einheiten Parfüm (P) und 2 Einheiten Zigaretten (C) schmuggeln. Der Profit für das Schmuggeln einer Einheit Whisky liegt bei 15$, der Profit für Parfüm und Zigaretten entsprechend bei 10$ und 7$. Was kann der Schmuggler in seinem Rucksack mitnehmen, wenn er min. 30$ Profit voraussetzt?

8.3.2 Initialisierung der Constraint-Netze

Zur Dokumentation des Testprogramms `YacsTester` erfolgt an dieser Stelle eine Übersicht über die erforderlichen Schritte, die für eine Initialisierung der Constraint-Netze bis zum Starten der Constraint-Auswertung erforderlich sind.[5]

Die `main()`-Routine von `YacsTester` (Zeile 519–521) erstellt eine Instanz der Klasse, in dessen Konstruktor die eigentlichen Testläufe durchgeführt werden (Zeile 441–517). Für jedes getestete Lösungsverfahren existiert eine eigene Methode, die vom Konstruktor jeweils aufgerufen wird.

Zunächst muss eine Instanz des YCM erzeugt werden. Dafür ist der Pfad zu den Constraint-Lösungsstrategien erforderlich. Es wird in diesem Fall lediglich der Default-Name verwendet (Zeile 41):

```
private final String strategyPath = "yacs_strategies.xml";
```

Die Instantiierung des Constraint-Managers wird in Zeile 447 von `YacsTester` vorgenommen:

```
this.ycm = new YacsConstraintManagerImpl(this.strategyPath);
```

Es werden als nächstes sämtliche Testroutinen initialisiert (Zeile 452–457). Während dieser Initialisierung werden für jeden Testfall eine Reihe von Constraints erzeugt und innerhalb eines String-Arrays gespeichert. Zusammen mit dem Namen der jeweils einzusetzenden Lösungsstrategie werden diese Arrays jeweils der Methode `initConstraints()` (Zeile 96) übergeben. Innerhalb dieser Methode werden in einer Iteration sämtliche in dem übergebenen Array enthaltenen Constraint-Ausdrücke zusammen mit dem Namen der jeweiligen Lösungsstrategie an den YCM übertragen (Zeile 99):

```
this.ycm.addConstraint(constraintArray[i], strategyName);
```

Diese Methode muss bereits innerhalb einer *try/catch*-Umgebung aufgerufen werden, um die ggf. vom YCM ausgelöste `ConstraintParserException` bzw. `StrategyNotFoundException` abfangen zu können. Während die erste Exception auf einen syntaktischen Fehler innerhalb des Constraint-Ausdrucks hinweist, muss letztere Exception bei jeder Gelegenheit abgefangen werden, wenn dem YCM der Name einer Strategie übergeben wird.

Anschließend müssen in jeder Testroutine die Wertebereiche der Constraint-Variablen belegt werden. Hierfür wird jeweils eine Liste angelegt, welche die Variablen der jeweiligen Strategie enthält:

```
variablesList = this.ycm.getVariables(strategyName);
```

Die von dieser Methode ggf. ausgelöste `StrategyNotFoundException` wird ebenso wie alle folgenden Exceptions in den Testroutinen abgefangen.

[5] Erwähnenswert ist, dass hierfür, wie auch im restlichen Programm, ausschließlich die „äußere Schnittstelle“ des YACS-Frameworks, d. h. die Methoden des Constraint-Managers, genutzt werden.

In den Testroutinen wird danach innerhalb einer Iteration jede Variable mit einer gültigen Domäne belegt. Beispielhaft für numerische finite Domänen wird an dieser Stelle die Erzeugung der Wertebereiche in `NCSolverTest` dargestellt (Zeile 136–137):

```
this.ycm.setVariableDomain((String)variablesList.get(i),
                           new NumericFDDomain(0, 9));
```

Neben numerischen finiten Domänen werden, speziell für das Kartenfärbeproblem in `AC3SolverTest`, auch symbolische finite Domänen erzeugt (Zeile 194–198):

```
domain = new SymbolicFDDomain();
domain.add(new SymbolicFDElement("rot"));
domain.add(new SymbolicFDElement("gruen"));
domain.add(new SymbolicFDElement("blau"));
this.ycm.setVariableDomain((String)variablesList.get(i), domain);
```

Die Testroutine für den `HullConsistencySolver` erstellt für jede der beteiligten Variablen eine Intervalldomäne (in diesem Fall ohne Iteration, Zeile 415–417):

```
this.ycm.setVariableDomain("v1", new IntervalDomain(2, 12));
this.ycm.setVariableDomain("v2", new IntervalDomain(4, 9));
this.ycm.setVariableDomain("v3", new IntervalDomain(2, 8));
```

Nach dieser Initialisierung befinden sich die jeweiligen Teilprobleme innerhalb von YACS in den Constraint-Netzen der zugehörigen Lösungsstrategie. Es erfolgen an dieser Stelle jeweils durch den Aufruf der Methode `printTestValues()` einige Ausgaben, um dem Nutzer die Überprüfung der Belegungen zu ermöglichen. Anschließend wird die Constraint-Auswertung gestartet werden (Zeile 465):

```
this.ycm.evaluate();
```

8.3.3 Ergebnisse des ersten Auswertevorgangs

Nach dem Durchlauf des Constraint-Lösungsvorgangs von YACS werden von `YacsTester` wiederum einige Ausgaben via `printTestValues()` getätigt, an denen die Veränderungen im Vergleich zu den vorherigen Ausgaben ersehen werden können. Aus Platzgründen können an dieser Stelle lediglich die Ausgaben *nach* der Constraint-Auswertung durch YACS wiedergegeben werden. Die genaue Problemdefinition ist bei Bedarf dem Programm-Listing von `YacsTester` in Anhang F zu entnehmen.

8.3.3.1 Knotenkonsistenz (1)

Zunächst wird an dieser Stelle die Ausgabe für die Strategie `low_consistency` betrachtet. Die FD-Variablen `x`, `y` und `z` wiesen zu Beginn jeweils eine Domäne mit den Werten von 0 bis 9 auf. Durch das Constraint `z<=2` müssen alle Werte aus der Domäne von `z` entfernt werden, die größer 2 sind. Das Constraint `3+y<7` bewirkt, dass aus der Domäne von `y` alle Werte entfernt werden, die größer 3 sind:

```
Ergebnis fuer Knotenkonsistenz:
===============================
Strategie: 'low_consistency'
Expression: (x<y); (y<z); (z<=2); ((3+y)<7)
Primitive Constraints: (4)
(x<y)
(y<z)
(z<=2)
((3+y)<7)
Variablen: [z, y, x]
Domaenen der Constraint-Variablen: (3)
z: [0, 1, 2]
y: [0, 1, 2, 3]
x: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Inkonsistenz: false

Ergebnis ist knotenkonsistent!
```

Nach der Propagation durch `NCSolver` wurden die Domänen von `z` und `y` entsprechend der Constraints eingeschränkt. Die Constraints `x<y` und `y<z` wurden nicht berücksichtigt, da es sich bei `NCSolver` lediglich um Knotenkonsistenz handelt. Das resultierende Constraint-Netz ist dementsprechend knotenkonsistent.

8.3.3.2 Kantenkonsistenz (1)

Für das Kartenfärbeproblem verfügen alle Variablen zu Beginn über eine Domäne mit den Elementen „rot", „gruen" und „blau". Während der Initialisierungsphase wurde allerdings eine Wertebereichseinschränkung vorgenommen, um eine Wertepropagation durch den Kantenkonsistenz-Algorithmus hervorzurufen: Die Domäne der Variable `WA` wurde auf die Farbe „rot" beschränkt (Zeile 202).[6] Entsprechend muss durch das Constraint `WA!=NT` das Element „rot" aus der Domäne von `NT` entfernt werden, da es (bidirektional ausgewertet) für diesen Wert keinen das Constraint erfüllenden Wert in der Domäne von `WA` mehr gibt. Analog gilt dies für das Constraint `WA!=SA` und die Domäne von `SA`. Das Ergebnis der Propagation stellt sich wie folgt dar:

```
Ergebnis fuer Kartenfaerbeproblem:
==================================
Strategie: 'medium_consistency'
Expression: (WA!=NT); (WA!=SA); (NT!=SA); (NT!=Q); (SA!=Q); (SA!=NSW);
            (SA!=V); (Q!=NSW); (T=T)
Primitive Constraints: (9)
(WA!=NT)
```

[6] Das für eine vollständige Beschreibung des Kartenfärbeproblems für eine Australienkarte fehlende Constraint `NSW!=V` wird für einen zweiten Auswertevorgang nachträglich hinzugefügt (vgl. Abschnitt 8.3.4, S. 233 und Abschnitt 8.3.5.2, S. 234).

```
(WA!=SA)
(NT!=SA)
(NT!=Q)
(SA!=Q)
(SA!=NSW)
(SA!=V)
(Q!=NSW)
(T=T)
Variablen: [NT, NSW, SA, WA, T, Q, V]
Domaenen der Constraint-Variablen: (7)
NT: [blau, gruen]
NSW: [blau, gruen, rot]
SA: [blau, gruen]
WA: [rot]
T: [blau, gruen, rot]
Q: [blau, gruen, rot]
V: [blau, gruen, rot]
Inkonsistenz: false

Ergebnis ist kantenkonsistent!
```

Im Ergebnis wurden die Wertebereiche von `NT` und `SA` korrekt entsprechend den oben genannten Bedingungen der Kantenkonsistenz eingeschränkt. Das Constraint-Netz ist weiterhin konsistent.

8.3.3.3 Einfache Suche (1)

Als nächstes erfolgt die Ausgabe der Strategie `simple_search`. Für die der Strategie zugeordneten Constraints `X<Y` und `Y<Z` existiert für die Variablen `X`, `Y` und `Z`, die jeweils über eine Domäne mit den Werten 1 und 2 verfügen, keine (globale) Belegung, die beide Constraints *gleichzeitig* erfüllt:

```
Ergebnis fuer Simple Search:
============================
Strategie: 'simple_search'
Expression: (X<Y); (Y<Z)
Primitive Constraints: (2)
(X<Y)
(Y<Z)
Variablen: [Z, Y, X]
Domaenen der Constraint-Variablen: (3)
Z: [1, 2]
Y: [1, 2]
X: [1, 2]
Inkonsistenz: true
```

```
Es konnte keine Loesung fuer das Problem ermittelt werden.
```

Korrekterweise hat der Suchalgorithmus keine Lösung für das Problem ermitteln können. Das Ergebnis enthält außerdem die Angabe, dass das zugehörige Constraint-Netz entsprechend als inkonsistent markiert wurde.

8.3.3.4 Vollständige Suche und Suche mit Look-Ahead (1)

Für das *Smuggler's-Knapsack*-Problem existieren vier mögliche Lösungen, die die beiden Constraints mit einer Begrenzung auf 9 Wareneinheiten und min. 30$ Profit erfüllen. Die Variablen `W` (Whisky), `C` (Zigaretten) und `P` (Parfüm) verfügen über Domänen mit den Werten von 0 bis 9. Eingesetzt in die beiden Ungleichungen

`(4*W)+(3*P)+(2*C)<=9` und `(15*W)+(10*P)+(7*C)>=30`,

ergeben sich ausschließlich die folgenden möglichen Belegungen:

1. `W = 0`, `C = 0`, `P = 3`
2. `W = 0`, `C = 3`, `P = 1`
3. `W = 1`, `C = 1`, `P = 1`
4. `W = 2`, `C = 0`, `P = 0`

Das Ergebnis der Strategie `search` stellt sich dementsprechend wie folgt dar:

```
Ergebnis fuer Smuggler's Knapsack (1):
======================================
Strategie: 'search'
Expression: ((((4*W)+(3*P))+(2*C))<=9); ((((15*W)+(10*P))+(7*C))>=30)
Primitive Constraints: (2)
((((4*W)+(3*P))+(2*C))<=9)
((((15*W)+(10*P))+(7*C))>=30)
Variablen: [W, C, P]
Domaenen der Constraint-Variablen: (3)
W: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
C: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
P: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Inkonsistenz: false

Loesung(en):
(1) W = 0, C = 0, P = 3;
(2) W = 0, C = 3, P = 1;
(3) W = 1, C = 1, P = 1;
(4) W = 2, C = 0, P = 0;
```

Der vollständige Suchalgorithmus der Strategie `search` liefert wie erforderlich alle möglichen Lösungen für das *Smuggler's-Knapsack*-Problem. Dies geschieht analog für die Strategie `search_with_look-ahead`, deren Ausgaben hier aus Platzgründen nicht separat aufgeführt werden.

8.3.3.5 Hull-Konsistenz (1)

Die Strategie mit dem Namen `interval_consistency` propagiert die vorliegenden *solution functions* entsprechend dem in Kapitel 5 ausführlich vorgestellten Beispiel 5.3.5 auf Seite 153. Ausgehend von den Wertebereichen [2,12] für die Intervallvariable `v1`, [4,9] für `v2` und [2,8] für `v3`, müssen im Ergebnis die Wertebereiche von `v1` und `v2` auf das Punktintervall [4,4] und der Wertebereich von `v3` auf das Punktintervall [8,8] beschränkt werden:

```
Ergebnis fuer Hull-Konsistenz:
==============================
Strategie: 'interval_consistency'
Expression: ((v1+v2)=v3); (v2=(v3-v1)); (v1=(v3-v2)); (v2=v1); (v1=v2)
Primitive Constraints: (5)
((v1+v2)=v3)
(v2=(v3-v1))
(v1=(v3-v2))
(v2=v1)
(v1=v2)
Variablen: [v1, v3, v2]
Domaenen der Constraint-Variablen: (3)
v1: [(4.0                +/- 8.881784197001252E-16) ]
v3: [(8.0                +/- 0.0                  ) ]
v2: [(4.0                +/- 8.881784197001252E-16) ]
Inkonsistenz: false

Ergebnis ist konsistent!
```

Die Einschränkungen der Intervalldomänen erfolgen analog zu dem in Abschnitt 5.3.5 auf Seite 153 ff. beschriebenen Beispiel. Die Einschränkungen erfolgten allerdings nicht soweit, wie es maximal möglich wäre. Stattdessen sind die Ergebnisintervalle geringfügig größer als erwartet, was mit der Unvollständigkeit von Berechnungen mit reellwertigen Intervallen durch die verwendete Intervallarithmetik-Bibliothek IAMath zu erklären ist.[7]

[7] Aufgrund der begrenzten Berechnungskapazität von Computern ist auch die IAMath-Bibliothek nur unvollständig – aber korrekt: Es ist sichergestellt, dass sich das gesuchte Ergebnis einer intervallarithmetischen Operationen innerhalb der zurückgegebenen Intervallgrenzen befindet. Dies bedingt allerdings, dass für manche Berechnungen geringfügig kleinere oder größere Ergebnisse für die untere bzw. obere Schranke zurückgegeben werden, als der Beobachter erwarten würde. Sind lediglich einfache Berechnungen notwendig und „aussagekräftige“ Ergebnisse erwünscht, wird man daher nicht um das Auf- bzw. Abrunden der oberen und unteren Schranken der Ergebnisintervalle umhin kommen.

8.3.4 Modifikation der Constraint-Netze

Im Anschluss an diesen ersten Auswertungsvorgang werden einige zusätzliche Constraints zu den vorhandenen Constraint-Netzen *inkrementell* hinzugefügt (Zeile 479–483):

```
this.ycm.addConstraint("z > 2;", "low_consistency");
this.ycm.addConstraint("NSW != V;", "medium_consistency");
this.ycm.addConstraint("W > 1;", "search");
this.ycm.addConstraint("W > 1;", "search_with_look-ahead");
this.ycm.addConstraint("v1 = [5,5];", "interval_consistency");
```

Außerdem werden an dieser Stelle einige Wertebereiche manuell modifiziert (Zeile 495, 497, 499):

```
this.ycm.setVariableDomain("V", new SymbolicFDDomain("blau"));
this.ycm.setVariableDomain("W", new NumericFDDomain(2));
this.ycm.setVariableDomain("Z", new NumericFDDomain(1, 3));
```

Nachdem die bestehenden Constraint-Netze innerhalb des YACS-Frameworks derart modifiziert wurden, erfolgt abschließend wiederum der Methodenaufruf zur Constraint-Auswertung (Zeile 509):

```
this.ycm.evaluate();
```

8.3.5 Ergebnisse des zweiten Auswertevorgangs

Die erneute Constraint-Auswertung führt zu weiteren Wertebereichseinschränkungen. Außerdem können, da zwischenzeitlich nicht nur neue Constraints inkrementell zu den Constraint-Netzen hinzugefügt sondern auch Domänen modifiziert wurden, für Constraint-Netze, deren Variablen nun über größere Wertebereiche verfügen, ggf. Lösungen generiert werden.

8.3.5.1 Knotenkonsistenz (2)

Nachdem der erneute Constraint-Auswertevorgang durchgeführt wurde, erfolgt wiederum die Ausgabe der nunmehr modifizierten Constraint-Netze. Dem Constraint-Netz der Strategie `low_consistency` wurde das Constraint `z>2` hinzugefügt. Zur Erfüllung dieses Constraints müssen entsprechend die verbleibenden Werte 0, 1 und 2 aus der Domäne von `z` entfernt werden, wodurch ein *domain wipe out* auftritt:

```
Ergebnis fuer Knotenkonsistenz:
===============================
Strategie: 'low_consistency'
Expression: (x<y); (y<z); (z<=2); ((3+y)<7); (z>2)
Primitive Constraints: (5)
(x<y)
```

```
(y<z)
(z<=2)
((3+y)<7)
(z>2)
Variablen: [z, y, x]
Domaenen der Constraint-Variablen: (3)
z: []
y: [0, 1, 2, 3]
x: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Inkonsistenz: true

Knotenkonsistenz nicht herstellbar!
```

Die Propagation ist korrekt und wie erwartet enthält die Domäne von `z` keinen Wert mehr. Knotenkonsistenz ist damit für dieses Constraint-Netz nicht mehr herstellbar. Das Constraint-Netz wird entsprechend als inkonsistent markiert.

8.3.5.2 Kantenkonsistenz (2)

Dem Constraint-Netz der Strategie `medium_consistency` wurde ebenfalls ein weiteres Constraint hinzugefügt: `NSW!=V`. Außerdem erfolgte für die Variable `V` die manuelle Einschränkung der zugehörigen Domäne auf das Element „blau“. Das Resultat dieser Einschränkungen ist ein (kanten-)inkonsistentes Constraint-Netz. Die Verarbeitung durch einen Algorithmus für Kantenkonsistenz muss eine Kettenreaktion auslösen, die dazu führt, dass der Wertebereich einer Variablen keine Elemente mehr enthält. Das Ergebnis der Strategie `medium_consistency` stellt sich wie folgt dar:

```
Ergebnis fuer Kartenfaerbeproblem:
==================================
Strategie: 'medium_consistency'
Expression: (WA!=NT); (WA!=SA); (NT!=SA); (NT!=Q); (SA!=Q); (SA!=NSW);
            (SA!=V); (Q!=NSW); (T=T); (NSW!=V)
Primitive Constraints: (10)
(WA!=NT)
(WA!=SA)
(NT!=SA)
(NT!=Q)
(SA!=Q)
(SA!=NSW)
(SA!=V)
(Q!=NSW)
(T=T)
(NSW!=V)
Variablen: [NT, NSW, SA, WA, T, Q, V]
Domaenen der Constraint-Variablen: (7)
```

```
NT: [blau]
NSW: [rot]
SA: [gruen]
WA: [rot]
T: [blau, gruen, rot]
Q: []
V: [blau]
Inkonsistenz: true

Kantenkonsistenz nicht herstellbar!
```

Die Propagation führt wie erwartet zu einem *domain wipe out*: Im Ergebnis existieren für die Variable `Q` keine konsistenten Belegungen in deren Wertbereich mehr, wodurch Kantenkonsistenz für das derart modifizierte Kartenfärbeproblem nicht mehr herstellbar ist. Das Constraint-Netz wird entsprechend als inkonsistent markiert.

8.3.5.3 Einfache Suche (2)

Für den zweiten Auswertevorgang wurde für die Strategie `simple_search` die Domäne der FD-Variable `Z` vergrößert: [1, 2, 3]. Für das Problem existiert nun mit der Belegung `X = 1`, `Y = 2` und `Z = 3` eine Lösung. Das Ergebnis der Auswertung durch die Strategie `simple_search` sieht wie folgt aus:

```
Ergebnis fuer Simple Search:
============================
Strategie: 'simple_search'
Expression: (X<Y); (Y<Z)
Primitive Constraints: (2)
(X<Y)
(Y<Z)
Variablen: [Z, Y, X]
Domaenen der Constraint-Variablen: (3)
Z: [1, 2, 3]
Y: [1, 2]
X: [1, 2]
Inkonsistenz: false

Loesung(en):
(1) Z = 3, Y = 2, X = 1;
```

Wie erwartet kann durch die Strategie `simple_search` für das modifizierte Constraint-Problem die Lösung generiert werden. Im Gegensatz zum Ergebnis des ersten Auswertevorgangs ist das Constraint-Netz daher entsprechend wieder als konsistent markiert.

8.3.5.4 Vollständige Suche (2)

Für das *Smuggler's-Knapsack*-Problem mit der Strategie `search` wurde ein weiteres Constraint `W>1` erzeugt. Außerdem wurde die Domäne der Variable `W` manuell auf den Wert 2 beschränkt. Damit verbleibt nur noch eine einzige Belegung als gültige Lösung für das derart modifizierte Constraint-Problem: `W=2`, `C=0` und `P=0`. Das Resultat der Strategie `search` sieht folgendermaßen aus:

```
Ergebnis fuer Smuggler's Knapsack (1):
======================================
Strategie: 'search'
Expression: ((((4*W)+(3*P))+(2*C))<=9); ((((15*W)+(10*P))+(7*C))>=30);
            (W>1)
Primitive Constraints: (3)
((((4*W)+(3*P))+(2*C))<=9)
((((15*W)+(10*P))+(7*C))>=30)
(W>1)
Variablen: [W, C, P]
Domaenen der Constraint-Variablen: (3)
W: [2]
C: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
P: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Inkonsistenz: false

Loesung(en):
(1) W = 2, C = 0, P = 0;
```

Im Ergebnis konnte die einzig verbliebende Lösung für das modifizierte *Smuggler's-Knapsack*-Problem durch den Suchalgorithmus gefunden werden.

8.3.5.5 Vollständige Suche mit Look-Ahead (2)

Zum *Smuggler's-Knapsack*-Problem mit der Strategie `search_with_look-ahead` wurde ebenfalls ein Constraint `W>1` hinzugefügt. Ersichtlich wird, dass, begründet durch den globalen Namensraum für Constraint-Variablen innerhalb von YACS, die Wertebereichseinschränkung für die Variable `W` auch diese Strategie beeinflusst. Entsprechend verbleibt auch für die Strategie `search_with_look-ahead` nur eine einzige Lösung für das Constraint-Problem: `W=2`, `C=0` und `P=0`. Das Ergebnis des Lösungsalgorithmus stellt sich wie folgt dar:

```
Ergebnis fuer Smuggler's Knapsack (2):
=======================================
Strategie: 'search_with_look-ahead'
Expression: ((((4*W)+(3*P))+(2*C))<=9); ((((15*W)+(10*P))+(7*C))>=30);
            (W>1)
```

```
Primitive Constraints: (3)
((((4*W)+(3*P))+(2*C))<=9)
((((15*W)+(10*P))+(7*C))>=30)
(W>1)
Variablen: [W, C, P]
Domaenen der Constraint-Variablen: (3)
W: [2]
C: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
P: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Inkonsistenz: false

Loesung(en):
(1) W = 2, C = 0, P = 0;
```

Die einzige Lösung für das Constraint-Problem konnte wie erwartet auch durch die Strategie `search_with_look-ahead` generiert werden.

8.3.5.6 Hull-Konsistenz (2)

Dem der Strategie `interval_consistency` zugeordneten intervallwertigen Problem wurde ein weiteres Constraint `v1=[5,5]` hinzugefügt. Dies kommt einer Wertebereichseinschränkung gleich und bewirkt, dass das Constraint-Problem inkonsistent ist. Die Ausgaben der Constraint-Verarbeitung durch die Strategie `interval_consistency` sieht folgendermaßen aus:

```
Ergebnis fuer Hull-Konsistenz:
==============================
Strategie: 'interval_consistency'
Expression: ((v1+v2)=v3); (v2=(v3-v1)); (v1=(v3-v2)); (v2=v1); (v1=v2);
            (v1=(5.0                +/- 0.0                ) )
Primitive Constraints: (6)
((v1+v2)=v3)
(v2=(v3-v1))
(v1=(v3-v2))
(v2=v1)
(v1=v2)
(v1=(5.0                +/- 0.0                ) )
Variablen: [v1, v3, v2]
Domaenen der Constraint-Variablen: (3)
v1: []
v3: [(8.0                +/- 0.0                ) ]
v2: [(4.0                +/- 8.881784197001252E-16) ]
Inkonsistenz: true

Konsistenz nicht herstellbar!
```

Durch den Constraint-Lösungsalgorithmus der Strategie `interval_consistency` wurde korrekterweise eine Inkonsistenz festgestellt. Dieses Problem ist damit nicht mehr lösbar. Das Constraint-Netz ist entsprechend als inkonsistent markiert.

8.3.6 Zusammenfassung

Durch das Programm `YacsTester` konnte gezeigt werden, dass sich mit dem YACS-Framework unterschiedliche Problemstellungen unabhängig voneinander mit unterschiedlichen Constraint-Lösungsstrategien verarbeiten lassen. Dabei können verschiedene Domänen bearbeitet werden: sowohl numerische und symbolische finite Domänen als auch intervallwertige infinite Domänen.

Das YACS-Framework bietet unterschiedliche Konsistenz- und Suchalgorithmen zur Auflösung von Constraint-Problemen. Constraints lassen sich inkrementell dem Lösungsprozess zuführen. Ebenso können Wertebereichseinschränkungen jederzeit vorgenommen werden. Der Constraint-Auswertevorgang kann nach solchen Aktionen wiederholt durchgeführt werden.

Anhand des Programms `YacsTester` wurde die Nutzung der „äußeren Schnittstelle“ (vgl. Abschnitt 7.9.2, S. 221) des YACS-Frameworks demonstriert. Die für die Initialisierung, Auswertung und Rückgabe der Ergebnisse der Constraint-Probleme benötigten Methoden werden für den Benutzer übersichtlich im Constraint-Manager von YACS zur Verfügung gestellt. Das Programm `YacsTester` ist vollständig in Anhang F auf Seite 315 ff. dokumentiert.

Nach diesen Validierungen mit unterschiedlichen synthetischen Problemstellungen soll anhand der Konfigurierung mit EngCon die Praxistauglichkeit des YACS-Frameworks demonstriert werden.

8.4 Validierung im Praxiseinsatz

In diesem Abschnitt wird anhand der implementierten EngCon-Schnittstelle des YACS-Frameworks (vgl. Abschnitt 7.9.1, S. 220) gezeigt, wie mit Hilfe von YACS unterschiedliche Propagierungen während eines Konfigurierungsvorgangs mit EngCon durchgeführt werden können.

8.4.1 Modifikation der Wissensbasen

Damit das Konfigurierungswerkzeug EngCon in der Lage ist das YACS-Framework für die Constraint-Auswertung zu nutzen, ist neben dem Austausch der Schnittstellenklasse eine Modifikation der jeweils eingesetzten Wissensbasis notwendig. Die Änderungen sind grundsätzlich minimal und betreffen ausschließlich die an den Constraint-Solver zu übergebenden Strings, welche die Constraint-Ausdrücke beinhalten. Diese werden innerhalb der Wissensbasis um den Namen der Constraint-Lösungsstrategie ergänzt, welche für die Verarbeitung des jeweiligen Constraints zuständig ist. Eine vereinfachte Schnittstelle könnte vorsehen, dass diese Zuordnung statisch innerhalb der Wrapper-Klasse zum YACS-

Framework vorgenommen wird. Zu Demonstrationszwecken wurde die flexiblere Variante gewählt, wodurch allerdings die Modifikationen erforderlich werden.

Für die Validierung mit dem Konfigurierungswerkzeug EngCon wurden zwei bereits existierende Wissensbasen an das YACS-Framework angepasst: `PC.lisp` und `PC_Advanced.lisp`. Mit beiden Wissensbasen lässt sich beispielhaft eine PC-Konfigurierung durchführen. An dieser Stelle werden die hierfür benötigten Konzepte, die Constraints und die Modifikationen in `PC_Advanced.lisp` dokumentiert, da diese Wissensbasis die Änderungen von `PC.lisp` sinngemäß beinhaltet, und sich zudem das nachfolgende Beispiel einer Konfigurierung auf `PC_Advanced.lisp` bezieht. Im Folgenden werden aus Platzgründen ausschließlich die für die Constraint-Propagation relevanten Teile der Wissensbasis aufgeführt.

8.4.1.1 Beispiel: Taktfrequenzen

Neben der oben angesprochenen Modifikation des zu übergebenen Constraint-Ausdrucks werden speziell für die Wissensbasis `PC_Advanced.lisp` einige weitere Änderungen notwendig, damit die Constraints für die Beispielkonfigurierung die ihnen zugedachte Funktionalität erfüllen können. So wird bei dem Konzept mit Namen `Processor` die `FSB_Rate` von einem Intervallwert auf diskrete Werte umgestellt:

```
(def-do
    :name Processor
    :oberkonzept PC_Component
    :parameter ((Clock {450 466 500 533 550 600 650 700 733 750 800 850})
                (Type {'Sockel_7 'Slot_A 'Slot_1})
                (FSB_Rate {66 100 133})
                (L2_Cache {128 512})
                (Icon "cpu" (non-config true))
                (Price [160 578])))
```

Für den Parameter `(FSB_Rate {66 100 133})` lässt sich auf diesem Weg ein FD-Solver verwenden, was dem Funktions-Constraint-Beispiel und dem Constraint-Netz aus Abschnitt 3.6.3 auf Seite 39 f. entspricht. Dieselbe Modifikation wird für das Konzept `Mainboard` notwendig:

```
(def-do
    :name Mainboard
    :oberkonzept PC_Component
    :parameter ((Type {'Sockel_7 'Slot_A 'Slot_1})
                (AGP_Slot {0 1})
                (PCI_Slot [3 6])
                (ISA_Slot [0 2])
                (FSB_Rate {66 100 133})
                (Icon "mainboard" (non-config true))
                (Price 350)))
```

Das ebenfalls benötigte Konzept `Memory` verfügt in seiner Ursprungsfassung bereits über diskrete Werte für den Parameter `FSB_Rate`:

```
(def-do
    :name Memory
    :oberkonzept PC_Component
    :parameter ((Capacity {16 32 64 128 256}(default 128))
                (FSB_Rate {66 100 133}(default 133))
                (Icon "simm" (non-config true))
                (Price [64 560])))
```

Die jeweiligen Unterkonzepte der genannten Konzepte benötigen keine Anpassung, da hier bereits jeweils ein diskreter Wert vorliegt. Die eigentliche Constraint-Relation mit dem zugehörigen konzeptuellen Constraint stellt sich demnach wie folgt dar:

```
(def-constraint-relation
    :name           func_FSB_Rate
    :constraint-typ :funktion
    :externe-pins   (MB_FSB_Rate P_FSB_Rate S_FSB_Rate)
    :beschraenkungs-funktion "MB_FSB_Rate = P_FSB_Rate;
                              MB_FSB_Rate <= S_FSB_Rate;
                              P_FSB_Rate <= S_FSB_Rate;
                              # medium_consistency")

(def-konzeptuelles-constraint
    :name                   FSB_Rate
    :variablen-pattern-paare ((?m  :name Mainboard)
                              (?p  :name Processor)
                              (?s :name Memory))
    :constraint-aufrufe      ((func_FSB_Rate (?m FSB_Rate)
                                             (?p FSB_Rate)
                                             (?s FSB_Rate))))
```

Während das konzeptuelle Constraint unverändert ist, wird die Constraint-Relation von einem Tupel-Constraint hin zu einem Funktions-Constraint modifiziert. In dem Constraint-Ausdruck sind drei primitive Constraints enthalten, welche die in Abschnitt 3.6.3 auf Seite 39 f. beschriebene Abhängigkeit repräsentieren. Der Constraint-Ausdruck, bestehend aus einer Konjunktion mehrerer primitiver Constraint-Ausdrücke, kann vom YACS Constraint-Parser in einem Schritt gelesen werden. Abgetrennt durch „#“ als Trennzeichen, wird an die Wrapper-Klasse des YACS-Frameworks der Name der zur Constraint-Verarbeitung zu nutzenden Lösungsstrategie übergeben.

8.4.1.2 Beispiel: CD-ROM-Geschwindigkeit

Für ein intervallwertiges Constraint-Beispiel wird ebenfalls ein bestehendes Constraint modifiziert. So wird dem Konzept mit dem Namen `CD_Rom` für den Parameter `Speed` ein

Werteintervall von 24 bis 40, und dem Parameter `Transfer_Rate` ein Werteintervall von 3600 bis 6000 zugewiesen:

```
(def-do
    :name CD_Rom
    :oberkonzept PC_Component
    :parameter ((Speed [24 40] (default 32))
                (Transfer_Rate [3600 6000])
                (Icon "cdrom" (non-config true))
                (Price {100 200})))
```

Die entsprechenden Unterkonzepte werden ebenfalls modifiziert. Nachfolgend ist die Constraint-Relation sowie das konzeptuelle Constraint für die Berechnung der Übertragungsgeschwindigkeit eines CD-ROM-Laufwerks aufgeführt. Das Constraint entspricht dem in Abschnitt 3.6.2 auf Seite 36 ff. beschriebenen Beispiel für ein Funktions-Constraint (vgl. Abbildung 3.9, S. 37):

```
(def-constraint-relation
    :name                     func_CD_Rom
    :constraint-typ           :funktion
    :externe-pins             (A B)
    :beschraenkungs-funktion "A = B * [150,150];
                              B = A / [150,150];
                              # interval_consistency")

(def-konzeptuelles-constraint
    :name                     conc_CD_Rom
    :variablen-pattern-paare ((?c :name CD_Rom))
    :constraint-aufrufe       ((func_CD_Rom (?c Transfer_Rate)
                                            (?c Speed))))
```

Auch hier erfährt das konzeptuelle Constraint keinerlei Modifikation. Die Beschränkungsfunktion der Constraint-Relation hingegen wird entsprechend dem in YACS enthaltenen Intervall-Constraint-Solver angepasst, durch den sich ausschließlich vorgefertigte *solution functions* verarbeiten lassen. Außerdem ist auch hier durch „#" als Trennzeichen der Name der für dieses Constraint zu nutzenden Constraint-Lösungsstrategie spezifiziert. Es ist wiederum nur ein einziger Constraint-Ausdruck, in diesem Fall bestehend aus zwei primitiven Constraints, erforderlich, der vom Constraint-Parser des YACS-Frameworks in einem Schritt geparst wird.

8.4.1.3 Weitere Funktions-Constraints

Die Wissensbasis `PC_Advanced.lisp` enthält daneben zwei weitere Funktions-Constraints, welche für eine Konfigurierung unter Verwendung von YACS ebenfalls angepasst werden müssen. Die nachfolgende Constraint-Relation, die festlegt, dass für einen bestimmten

Prozessortyp das in der Konfiguration enthaltene Netzgerät des Computergehäuses eine erhöhte Leistungsabgabe aufweisen muss, wird lediglich um die zu nutzende Constraint-Lösungsstrategie erweitert. Das konzeptuelle Constraint erfährt keinerlei Modifikation:

```
(def-constraint-relation
    :name                   func_Power_Supply
    :constraint-typ         :funktion
    :externe-pins           (A)
    :beschraenkungs-funktion "A = 300; # low_consistency")

(def-konzeptuelles-constraint
    :name                   conc_Power_Supply
    :variablen-pattern-paare ((?v :name Processor
                                  :parameter((Type 'Slot_A)))
                              (?m :name Tower))
    :constraint-aufrufe     ((func_Power_Supply (?m power_supply))))
```

Da es sich hier lediglich um ein unäres Constraint handelt, ist die Constraint-Lösungsstrategie für Knotenkonsistenz mit dem Namen `low_consistency` ausreichend.

Das letzte zu modifizierende Funktions-Constraint dient dazu sicherzustellen, dass sich nach der Auswahl einer Grafikkarte mit AGP-Schnittstelle ausschließlich `Mainboard`-Konzepte in der Teilkonfiguration befinden, die einen entsprechenden AGP-Slot aufweisen. Das Constraint entspricht dem in Abschnitt 3.6.1 auf Seite 35 f. dargelegten Beispiel (vgl. Abbildung 3.8, S. 35):

```
(def-constraint-relation
    :name                   func_AGP_Mainboard
    :constraint-typ         :funktion
    :externe-pins           (A)
    :beschraenkungs-funktion "A = 1; # low_consistency")

(def-konzeptuelles-constraint
    :name                   conc_AGP_Mainboard
    :variablen-pattern-paare ((?v :name VGA_Card
                                  :parameter((Bus 'agp)))
                              (?m :name Mainboard))
    :constraint-aufrufe     ((func_AGP_Mainboard (?m AGP_Slot))))
```

Auch diese Constraint-Relation muss lediglich um die zu verwendende Constraint-Lösungsstrategie erweitert werden, wobei auch hier Knotenkonsistenz ausreichend ist. Das konzeptuelle Constraint erfährt keinerlei Modifikation.

In dem folgenden Abschnitt wird beispielhaft eine Konfigurierung mit den hier beschriebenen Constraints und dem Konfigurierungswerkzeug EngCon durchgeführt.

Pin	Value	C0	C1	C2	C3	C4	C5	C6	C7
Memory_7.FSB_Rate	66 100 133	X	X	X	X	X			
Processor_3.FSB_Rate	66 100 133					X			
Processor_3.Price	[160 .. 578]						X		
Memory_7.Price	[64 .. 560]						X		
CD_Rom_5.Speed	[24 .. 40]							X	
PC_0.RAM_check	0 1 2	X			X				
CD_Rom_5.Transfer_Rat	[3600 .. 6000]							X	
Mainboard_1.FSB_Rate	66 100 133		X	X		X			
Processor_3.Type	Sockel_7 Slot_...								X
PC_0.Price	[224.0 .. 1138.0]						X		
Mainboard_1.Type	Sockel_7 Slot_...								X

Abbildung 8.1: Das Constraint-Netz nach dem Start der Konfigurierung

8.4.2 Konfigurierung mit EngCon

Basierend auf der Wissensbasis `PC_Advanced.lisp`, den in dem vorhergehenden Abschnitt beschriebenen Modifikationen und der durch eine Wrapper-Klasse realisierten Anbindung des YACS-Frameworks an das Konfigurierungswerkzeug EngCon (vgl. Abschnitt 7.9.1, S. 220) erfolgt an dieser Stelle ein Ausschnitt aus einer beispielhaft durchgeführten Konfigurierung. Die nachfolgende Beschreibung konzentriert sich auf die Propagation der in der Wissensbasis spezifizierten Funktions-Constraints. Die Namen der relevanten Constraint-Variablen, Konzeptinstanzen und Parameter wurden in den Abbildungen jeweils hervorgehoben:

- Das Constraint-Netz nach dem Starten der Konfigurierung ist in Abbildung 8.1 dargestellt. Die Wertebereiche für den Parameter `FSB_Rate` der Konzeptinstanzen von `Memory`, `Processor` und `Mainboard` enthalten jeweils die Werte 66, 100 und 133.[8] Die Wertebereiche für die Parameter `Transfer_Rate` und `Speed` der Konzeptinstanz von `CD_Rom` sind mit Intervallen von 3600 bis 6000 bzw. von 24 bis 40 initialisiert.

- In Abbildung 8.2 auf der nächsten Seite ist dargestellt, wie durch den Benutzer innerhalb der grafischen Benutzungsoberfläche von EngCon ein Prozessor spezialisiert wird. Die Auswirkungen der Auswahl des Prozessortyps „Athlon“ durch den Benutzer sind in Abbildung 8.3 auf der nächsten Seite zu sehen: Die Parameter `FSB_Rate` für die Konzeptinstanzen von `Processor` und `Mainboard` wurden von YACS entsprechend der in Abschnitt 8.4.1.1 auf Seite 240 definierten Constraint-Relation `func_FSB_Rate` auf den Wert 100 eingeschränkt. Der Parameter `FSB_Rate` für die Konzeptinstanz von `Memory` wird entsprechend auf die Werte 100 und 133 beschränkt. Außerdem wurde die unäre Constraint-Relation `func_Power_Supply` aktiviert und dem Constraint-Netz *inkrementell* hinzugefügt. Dieses Constraint stellt sicher, dass ein Computergehäuse mit einem Netzgerät mit entsprechender Leistungsabgabe Verwendung findet (vgl. Abschnitt 8.4.1.3, S. 242).

[8]Die Nummerierung der Konzeptinstanzen (z. B. `Memory_7`) wird in dieser Beschreibung nicht berücksichtigt.

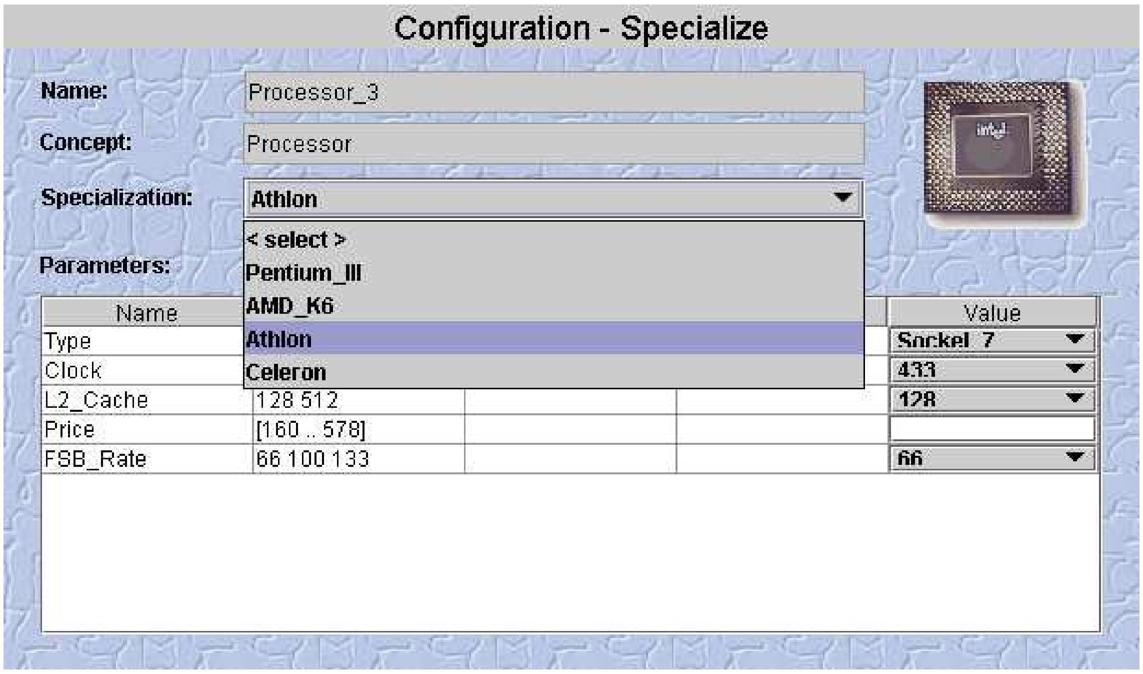
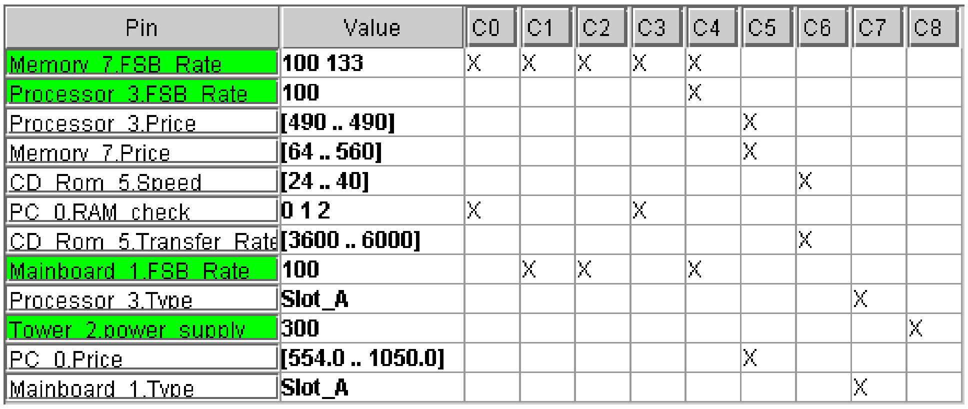


Abbildung 8.2: Spezialisierung der Konzeptinstanz von `Prozessor`



Pin	Value	C0	C1	C2	C3	C4	C5	C6	C7	C8
Memory_7.FSB_Rate	100 133	X	X	X	X	X				
Processor_3.FSB_Rate	100					X				
Processor_3.Price	[490 .. 490]						X			
Memory_7.Price	[64 .. 560]						X			
CD_Rom_5.Speed	[24 .. 40]							X		
PC_0.RAM_check	0 1 2	X			X					
CD_Rom_5.Transfer_Rate	[3600 .. 6000]							X		
Mainboard_1.FSB_Rate	100		X	X		X				
Processor_3.Type	Slot_A								X	
Tower_2.power_supply	300									X
PC_0.Price	[554.0 .. 1050.0]						X			
Mainboard_1.Type	Slot_A								X	

Abbildung 8.3: Einschränkung der Parameter `FSB_Rate`

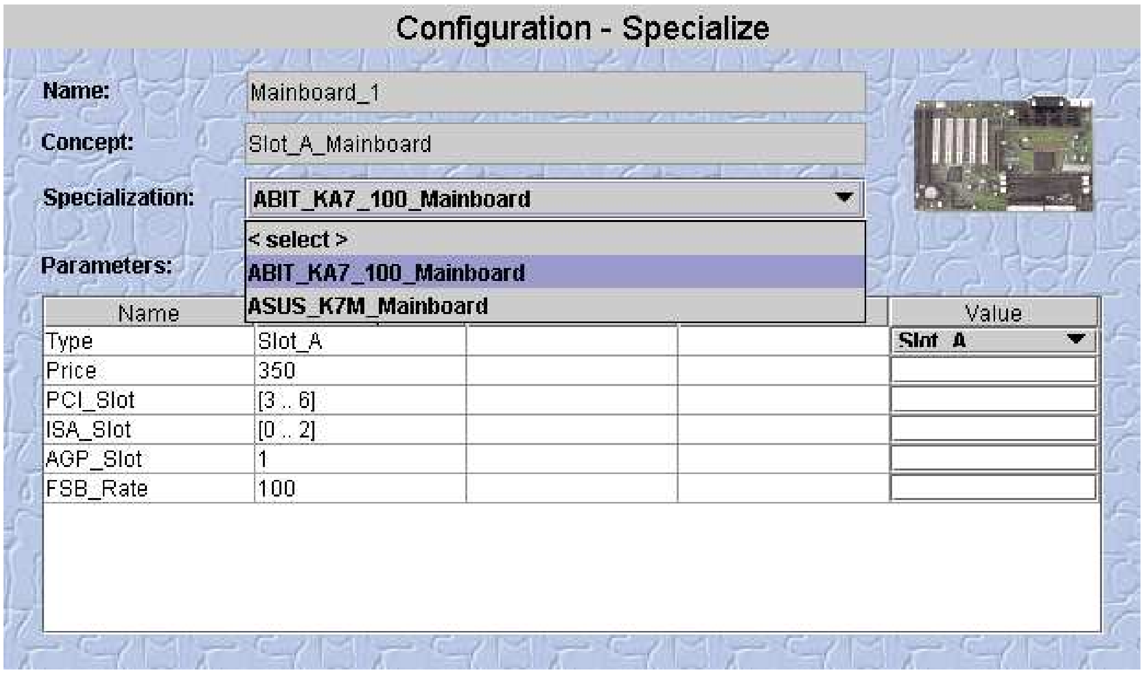


Abbildung 8.4: Spezialisierung der Konzeptinstanz von `Mainboard`

- Durch die Spezialisierung der Konzeptinstanz von `Mainboard`, dargestellt in Abbildung 8.4, werden keine weiteren Wertebereichseinschränkungen verursacht, allerdings wurde durch die vorherige Auswahl einer Grafikkarte mit AGP-Schnittstelle die Constraint-Relation `func_AGP_Mainboard` *inkrementell* dem Constraint-Netz hinzugefügt (vgl. Abbildung 8.5 auf der nächsten Seite). Die Constraint-Relation `func_AGP_Mainboard` stellt sicher, dass für die Spezialisierung der Konzeptinstanz von `Mainboard` ausschließlich Mainboards zur Auswahl stehen, deren Parameter `AGP_Slot` mit dem Wert 1 belegt ist (vgl. Abschnitt 8.4.1.3, S. 242).

- In der Abbildung 8.6 auf der nächsten Seite ist zu sehen, wie der Benutzer in EngCon eine Parametrierung der Konzeptinstanz von `Memory` durchführt. Die Auswahl des Werts 133 hat zur Folge, dass das Constraint-Netz bzgl. der Parameter `FSB_Rate` seinen Endzustand erreicht: Der Parameter `FSB_Rate` für die Konzeptinstanz von `Memory` wird auf den Wert 133 beschränkt. Für die Konzeptinstanzen von `Processor` und `Mainboard` bleibt die Belegung des Parameters `FSB_Rate` jeweils entsprechend der Constraint-Relation `func_FSB_Rate` mit dem Wert 100 bestehen. Die Konsistenz des Constraint-Netzes ist weiterhin gewährleistet (vgl. Abbildung 8.7 auf Seite 247).

- Abschließend erfolgt die Parametrierung der gewünschten Übertragungsgeschwindigkeit des auszuwählenden CD-ROM-Laufwerks. In Abbildung 8.8 auf Seite 248 trägt der Benutzer den Wert 4800 für den Parameter `Transfer_Rate` der Konzeptinstanz von `CD_Rom` ein. Die darauf folgend einsetzende Propagation innerhalb von YACS berechnet anhand der Constraint-Relation `func_CD_Rom` (vgl. Abschnitt 8.4.1.2, S. 241)

Pin	Value	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
Processor_3.Price	[490 .. 490]	X									
Processor_3.FSB_Rate	100		X								
CD_Rom_5.Speed	[24 .. 32]			X							
PC_0.RAM_check	0 1 2				X	X					
CD_Rom_5.Transfer_Rate	[3600 .. 4800]			X							
Mainboard_1.Type	Slot_A						X				
Memory_7.FSB_Rate	100 133		X		X	X		X	X		
Memory_7.Price	[64 .. 560]	X									
Mainboard_1.FSB_Rate	100		X					X	X		
Processor_3.Type	Slot_A						X				
Tower_2.power_supply	300									X	
Mainboard_1.AGP_Slot	1										X
PC_0.Price	[554.0 .. 1050.0]	X									

Abbildung 8.5: Die Constraint-Relation `func_AGP_Mainboard`

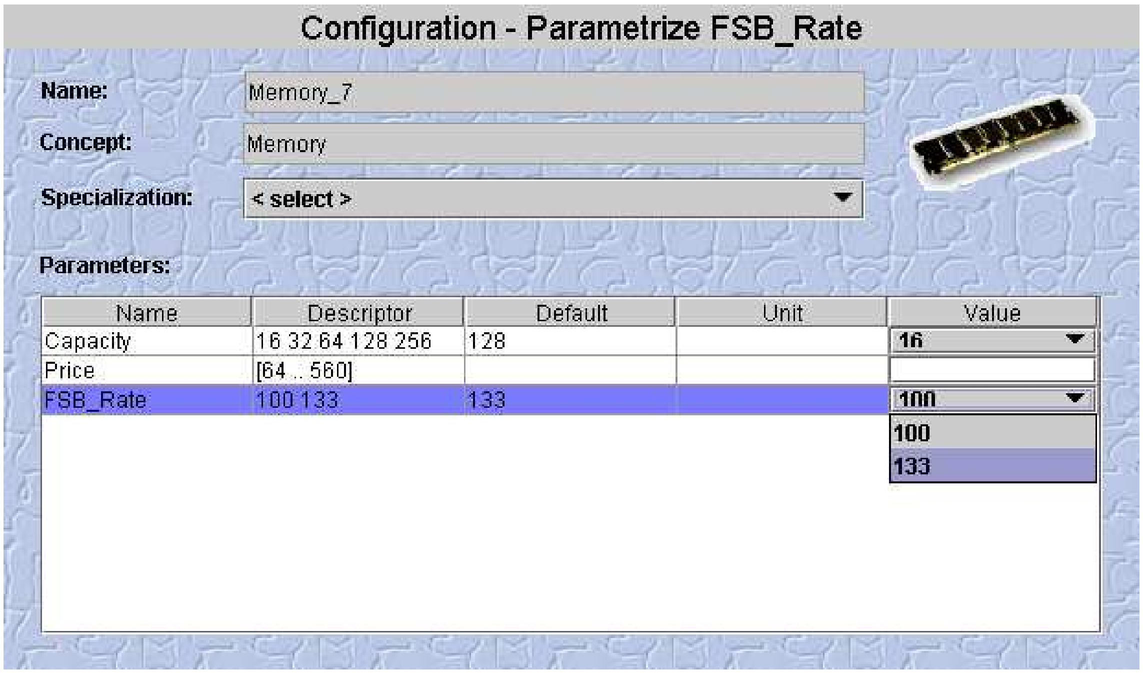


Abbildung 8.6: Parametrierung von `FSB_Rate` der Konzeptinstanz von `Memory`

Pin	Value	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
Processor_3.Price	[490 .. 490]	X									
Processor_3.FSB_Rate	100		X								
CD_Rom_5.Speed	[24 .. 32]			X							
PC_0.RAM_check	2				X	X					
CD_Rom_5.Transfer_Rate	[3600 .. 4800]			X							
Mainboard_1.Type	Slot_A						X				
Memory_7.FSB_Rate	133		X		X	X		X	X		
Memory_7.Price	[64 .. 560]	X									
Mainboard_1.FSB_Rate	100		X					X	X		
Processor_3.Type	Slot_A						X				
Tower_2.power_supply	300									X	
Mainboard_1.AGP_Slot	1										X
PC_0.Price	[554.0 .. 1050.0]	X									

Abbildung 8.7: Erneute Einschränkung des Parameters `FSB_Rate`

den korrelierenden Wert 32 für den Parameter `Speed` derselben Konzeptinstanz von `CD_Rom` und schränkt dessen Wertebereich entsprechend ein. Das Constraint-Netz in seiner endgültigen Ausprägung ist in Abbildung 8.9 auf der nächsten Seite dargestellt.

Die Konfigurierung ist damit beendet. Nicht berücksichtigt in dieser Übersicht wurden Konfigurierungsschritte abseits des Constraint-Systems von EngCon bzw. Konfigurierungsschritte, die unabhängig von YACS durchgeführt wurden.

8.4.3 Zusammenfassung

Anhand dieses Ausschnitts aus dem praktischen Einsatz wurde gezeigt, wie durch die wiederholte Propagation der in Abschnitt 8.4.1 auf Seite 238 ff. vorgestellten Funktions-Constraints die wissensbasierte Konfigurierung mit EngCon von YACS unterstützt werden kann. Der Zugriff auf das YACS-Framework erfolgt über die bestehende Schnittstelle von EngCon über eine Wrapper-Klasse (vgl. Abschnitt 7.9.1, S. 220). Es lassen sich flexibel unterschiedliche Constraint-Solver nutzen, sowohl für Constraints mit finiten als auch mit infiniten Domänen. Constraints lassen sich inkrementell den Constraint-Netzen zuführen. Wertebereichseinschränkungen innerhalb der Konfigurierung mit EngCon werden ebenfalls problemlos in die auszuwertenden Constraint-Netze übernommen.

Für eine Nutzung der bestehenden Wissensbasen von EngCon sind selbige geringfügig anzupassen. Dies geschieht zugunsten der Flexibilität, damit sich für unterschiedliche Problemstellungen innerhalb einer Wissensbasis unterschiedliche Constraint-Solver einsetzen lassen. Eine Alternative bestünde darin, die Auswahl eines statischen Constraint-Solvers in der Wrapper-Klasse zu implementieren. Für die EngCon-Schnittstelle gelten außerdem die in Abschnitt 7.9.1 auf Seite 220 ff. angesprochenen Einschränkungen.

Nachfolgend wird im Rahmen dieser Validierung ein Vergleich des Konzepts für das YACS-Framework mit weiterführenden Arbeiten vorgenommen.

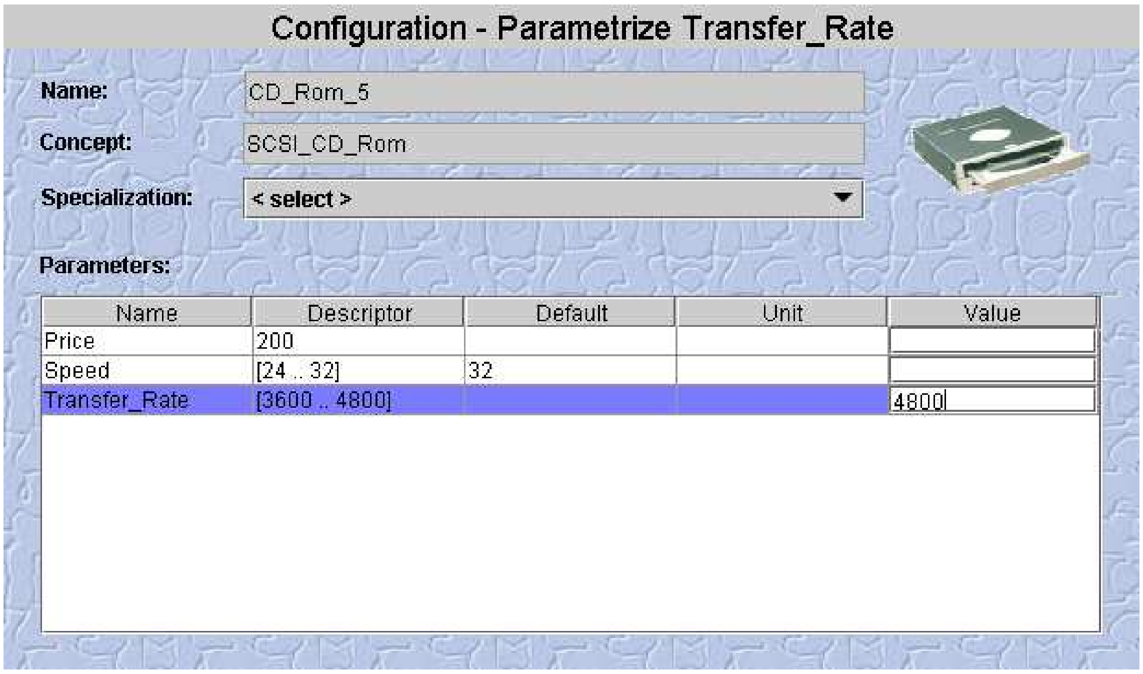


Abbildung 8.8: Parametrierung von `Transfer_Rate` der Konzeptinstanz von `CD_Rom`

Pin	Value	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
Processor_3.Price	**[490 .. 490]**	X									
Processor_3.FSB_Rate	**100**		X								
CD_Rom_5.Speed	**32**			X							
PC_0.RAM_check	**2**				X	X					
CD_Rom_5.Transfer_Rate	**[4800 .. 4800]**			X							
Mainboard_1.Type	**Slot_A**						X				
Memory_7.FSB_Rate	**133**		X		X	X		X	X		
Memory_7.Price	**[64 .. 560]**	X									
Mainboard_1.FSB_Rate	**100**		X					X	X		
Processor_3.Type	**Slot_A**						X				
Tower_2.power_supply	**300**									X	
Mainboard_1.AGP_Slot	**1**										X
PC_0.Price	**[554.0 .. 1050.0]**	X									

Abbildung 8.9: Endgültig propagiertes und konsistentes Constraint-Netz

8.5 Vergleich mit weiterführenden Ansätzen

Im Zuge der Konzeption von YACS wuchs die Erkenntnis, dass sich innerhalb eines hybriden Systems nicht nur die flexible Kombination unterschiedlicher Constraint-Lösungsmethoden betrachten lässt, sondern auch die flexible Kooperation eben dieser Verfahren bzw. Constraint-Solver. Zur Erhöhung der Flexibilität ist die Anwendung strategiebasierter Ansätze ein gängiges Mittel zur Steuerung der Propagation und Lösungssuche von kooperierenden Constraint-Solvern. Der konkrete Einsatz von Strategien in den vorhandenen Ansätzen unterscheidet sich allerdings von der spezifischen Nutzung innerhalb von YACS, daher erfolgt an dieser Stelle insbesondere ein Vergleich mit Ansätzen zur flexiblen Kooperation.

In Abschnitt 4.5.4 auf Seite 77 ff. wurde bereits eine Übersicht über verschiedene Ansätze für kooperatives Verhalten von Constraint-Solvern gegeben. Es wird grundsätzlich zwischen statischen „Ad-hoc"-Kooperationen und flexiblen Kooperationen unterschieden. Die genannten Ansätze weisen z. T. eine enge Verwandtschaft mit Distributed CSP bzw. CCP auf (vgl. Abschnitt 4.4.4, S. 62). Eine verteilte Ausführung allerdings wird von YACS, bedingt durch ein sequentielles anstatt paralleles Ausführungsmodell, derzeit nicht explizit unterstützt.

Im Vergleich zu Ad-hoc-Kooperationen bietet das YACS-Framework flexible Kooperationen mit aufeinander aufbauenden und austauschbaren Lösungsverfahren. Kooperationen auf Algorithmusebene können innerhalb eines Constraint-Solvers realisiert werden. Dafür lassen sich bereits existierende Algorithmen auf Implementierungsebene zu kooperierenden Constraint-Solvern zusammenführen (z. B. für *look-ahead*, vgl. Abschnitt 7.6, S. 209). Außerdem ist mit dem YACS-Framework derzeit eine hybride Constraint-Verarbeitung, d. h. unterschiedliche und auch erweiterbare Domänen, möglich, eine heterogene Verarbeitung von mixed Constraints (vgl. Abschnitt 4.4.4, S. 64) wird hingegen noch nicht unterstützt.

Die Systeme zur flexiblen Kooperation von Hofstedt (2000) und Monfroy (2000) nutzen formale Ansätze bzw. Sprachen zur Definition von Ausführungsvorschriften respektive zur Definition von Kooperationsstrategien. Dadurch weisen beide Ansätze einen hohen Freiheitsgrad bzgl. der definierbaren Kooperationen auf. Im Vergleich dazu ist das Ausführungsmodell innerhalb von YACS relativ fest vorgegeben. Es kann lediglich gewählt werden, welche Lösungskomponenten für die jeweilige Kooperation in welcher Reihenfolge genutzt werden sollen. Die hierfür benötigte Spezifikation von Constraint-Lösungsstrategien wird anstatt in einer eigens entwickelten Sprache in XML-Syntax vorgenommen, und anstatt einer Grammatik für eine eigene Sprache liegt zur Beschreibung von Constraint-Lösungsstrategien für YACS eine XML-DTD vor. YACS weist somit durch Nutzung der verbreiteten XML-Technologie einen hohen Praxisbezug auf und ist relativ einfach einsetzbar und erweiterbar. Darüber hinaus ist der Integrationsaufwand für YACS, speziell für Java-Anwendungen, deutlich geringer als für Lösungen, die in Lisp, ECL^{i}PS^e oder Manifold implementiert wurden (vgl. Abschnitt 4.5.4.2, S. 77).

Die innerhalb von YACS vorgenommene Unterteilung des Lösungsvorgangs in einzelne Phasen führt zu einer „Strukturierung" des Lösungsprozesses. Der Vorteil besteht darin, dass für jede Phase spezielle Schnittstellen im System vorgesehen werden können. Dadurch wiederum wird der flexible Austausch der Constraint-Solver ermöglicht, die jeweils eine einheitliche Schnittstelle entsprechend der jeweiligen Phase aufweisen müssen. Der phasenweise Lösungsprozess ermöglicht außerdem die Ausnutzung von Seiteneffekten während einer Phase. Wertebereichseinschränkungen unterschiedlicher Constraint-Solver lassen sich dementsprechend teilproblemübergreifend propagieren bis keinerlei Einschränkungen mehr auftreten. Insgesamt kann so i. A. eine erhöhte Filterstärke erreicht werden. Nachteilig am vorgestellten Konzept von YACS kann sich ggf. auswirken, dass die Einteilung in die bestehenden Phasen fest vorgegeben und damit implementierungsabhängig relativ starr ist. Für einzubindende, externe Constraint-Solver ist zudem jeweils eine spezielle Wrapper-Klasse für die entsprechende Schnittstelle der jeweiligen Phase zu implementieren, was sich negativ auf den zusätzlichen Overhead auswirkt.[9]

Die Möglichkeit, auf unkomplizierte und durchschaubare Art und Weise Anpassungen an YACS bzw. an den Constraint-Lösungsstrategien von YACS vornehmen zu können, kann sich positiv auf die Akzeptanz der Lösung bei den zuständigen Wissensingenieuren bei gleichzeitig höchstmöglicher Flexibilität auswirken. Das YACS-Framework wurde zudem für den Anwendungsfall der strukturbasierten Konfigurierung entwickelt und getestet. YACS unterstützt daher die domänenspezifischen Eigenheiten wie ein inkrementell anwachsendes Constraint-Netz und sowohl finite als auch infinite Wertebereiche.

Das Konzept für YACS wurde aus der Idee heraus geboren, größtmögliche Flexibilität hinsichtlich der einzusetzenden Constraint-Lösungsmechanismen zu bieten. Weniger im Vordergrund steht der Aspekt der flexiblen *Steuerung* von Kooperationen unterschiedlicher Lösungsverfahren. Die Art der Kooperation ist innerhalb von YACS grundsätzlich durch den Aufbau der Constraint-Lösungsstrategien und deren bisher ausschließlich *sequentiellen* Ausführung vorgegeben. Flexibilität wird innerhalb von YACS durch unterschiedliche und auswechselbare Constraint-Solver für finite und infinite Domänen sichergestellt. Auch wenn mit YACS durchaus flexible Kooperationen mit austauschbaren, aufeinander aufbauenden Constraint-Solvern und unterschiedlichen Domänen möglich sind, stellt das YACS-Framework einen Mittelweg dar zwischen einer flexiblen Kooperation, wie sie die Kooperationssprachen von Hofstedt (2000) und Monfroy (2000) bieten, und einer starren Ad-hoc-Kooperation.

8.6 Zusammenfassung

Im Rahmen dieser Validierung wurde die Praxistauglichkeit des dem Prototypen von YACS zugrundeliegenden Konzepts unter Beweis gestellt. Anhand einer Reihe von synthetischen Problemen sowie der Integration in das Konfigurierungswerkzeug EngCon konnte gezeigt werden, dass sich mit Hilfe des YACS-Frameworks in verschiedenen Problemstellungen unterschiedliche Constraint-Lösungsverfahren flexibel einsetzten lassen. Überlappungen von Teilproblemen sind hierbei möglich, es muss allerdings sichergestellt sein, dass die von den

[9] Für externe Constraint-Solver bedeutet die Anbindung über das YACS-Framework eine zusätzliche Repräsentation des Constraint-Netzes: Neben der Repräsentation in der Anwendung besitzt YACS eine eigene, interne Repräsentation der Constraints. Externe Constraint-Solver, die die interne Repräsentation von YACS nicht nutzen, benötigen folglich eine Konvertierung und zusätzlichen Speicher für ihre eigene Repräsentation.

Überlappungen betroffenen Constraint-Netze jeweils dieselben Wertedomänen aufweisen (numerische finite Domänen, symbolische finite Domänen oder reellwertige Intervalldomänen).

Innerhalb des YACS-Frameworks steht für die effiziente Verarbeitung von Constraint-Problemen eine überschaubare Bibliothek von Constraint-Solvern zur Verfügung. Für unterschiedliche Anwendungsdomänen kann ein jeweils geeigneter Constraint-Solver eingesetzt werden. Auch der Einsatz unterschiedlicher Constraint-Solver in derselben Domäne ist bei Bedarf möglich. Die tatsächliche Effizienz der Constraint-Lösungsverfahren trat allerdings bei der Umsetzung des Konzepts für das YACS-Framework in den Hintergrund. Der Schwerpunkt der Realisierung von YACS liegt auf der Flexibilität und der Modularität des Ansatzes.

Die von YACS zur Verfügung gestellten Constraint-Solver arbeiten als *Black Box* und besitzen keinen Einfluss auf die Ablaufkontrolle des Frameworks. Sie sind gekapselt und austauschbar. Fremdsysteme sind über Wrapper-Klassen in das YACS-Framework integrierbar. Besonders die stringbasierte Schnittstelle von YACS, zur intensionalen Repräsentation von Constraints, ist ein interessantes Feature für ein breites Anwendungsfeld. Durch diese Schnittstelle wird die komfortable Nutzung von Constraint-Solvern als Inferenzmechanismus ermöglicht. Anwendungen, die diese Schnittstelle nicht nutzen, bleibt der Weg über die internen Klassen des Frameworks. Über diese ist das YACS-Framework zudem erweiterbar, z. B. um zusätzliche Constraint-Solver oder Variablentypen bzw. Wertedomänen. Eine Erweiterung von YACS kann durch Einbinden zusätzlicher *Java-Archives* (JARs) geschehen. Die in diesen JARs enthaltenen Java-Klassen haben lediglich die von dem Framework vorgegebenen Vererbungshierarchien und Paketstrukturen einzuhalten, sowie die ebenfalls von den Framework-Klassen definierten Schnittstellen zu erfüllen. Zusätzliche Constraint-Solver und andere Erweiterungen können auf diesem Weg ähnlich einem Plugin-Mechanismus ergänzt werden.

Wesentliche Anforderung durch den in dieser Arbeit gegebenen Anwendungsfall, der strukturbasierten Konfigurierung, ist der inkrementelle Aufbau des Constraint-Netzes. Diese Anforderung konnte erfüllt werden, wie die beiden Validierungen in diesem Kapitel gezeigt haben: Sowohl bei den synthetischen Constraint-Problemstellungen als auch beim Anwendungsfall EngCon ist das inkrementelle Anwachsen des Constraint-Netzes zu beobachten. Für die synthetischen Constraint-Probleme konnte ferner gezeigt werden, dass die hinzugefügten Constraints konkrete Auswirkungen auf die bestehenden Wertebereiche der Variablen bereits existierender Constraints haben.

Eine weitere, wesentliche Anforderung sind die unterstützten Wertedomänen der Lösungsverfahren. Das YACS-Framework ist ein hybrides System, mit dem sich sowohl finite als auch infinite Domänen verarbeiten lassen. Neben numerischen und symbolischen finiten Domänen werden durch das YACS-Framework derzeit reellwertige Intervalldomänen unterstützt. Es sind allerdings ausschließlich konvexe Intervalldomänen zulässig, zudem ist der Intervall-Constraint-Solver von YACS derzeit grundsätzlich noch mit starken Einschränkungen behaftet. Weitere geforderte Eigenschaften an die Lösungsverfahren, wie die Möglichkeit zur Verarbeitung n-ärer und nichtlinearer Constraints werden vom YACS-Framework zurzeit rudimentär angeboten: Derartige Constraints werden von den implementierten Lösungsverfahren bereits unterstützt, es existieren allerdings noch keine spezialisierten Algorithmen innerhalb der Constraint-Solver-Bibliothek von YACS.

Den Anforderungen an die Präzision und Effizienz des entwickelten Constraint-Systems wird mit der Möglichkeit der flexiblen Austauschbarkeit der einzusetzenden Lösungsverfahren Rechnung getragen. Bei Bedarf lassen sich jederzeit effizientere Verfahren als die derzeit vorhandenen Lösungsverfahren implementieren und einbinden. Die Vollständigkeit der zurzeit vorhandenen Lösungsverfahren ist sichergestellt, mögliche Lösungen gehen bei den implementierten Verfahren wie Backtracking, MAC, Knoten-, Kanten- und Hullkonsistenz auf keinen Fall verloren. Eine Ausnahme bildet der Constraint-Solver für einfaches Backtracking, der lediglich die erstbeste Lösung zurückliefert. Dieser Solver ist für spezielle Probleme geeignet, wenn es bspw. darum geht, die Inkonsistenz eines Problems festzustellen.

Die Berechnungen, die von der innerhalb von YACS eingebundenen Intervallarithmetik-Bibliothek IAMath durchgeführt werden, sind aufgrund der begrenzten Berechnungskapazität von Computern grundsätzlich unvollständig. Dies äußert sich allerdings in diesem Fall nicht darin, dass Werte der Lösungsmenge verloren gehen, sondern indem die Intervallgrenzen bei bestimmten Berechnungen in der Praxis nicht so stark eingeschränkt werden, wie dies theoretisch möglich ist.[10] Für Anwendungen, für die die Präzision der verwendeten Intervallarithmetik-Bibliothek nicht ausreicht, bieten sich die Möglichkeiten, eine eigene Bibliothek zu entwickeln oder eine andere, externe Bibliothek anzubinden. Mit der Kapselung des Zugriffs durch Framework-Klassen ist sichergestellt, dass die derzeit verwendete Bibliothek mit minimalem Aufwand austauschbar ist bzw. das YACS-Framework um weitere Arithmetik-Bibliotheken ergänzt werden kann.[11]

Die Implementierung von YACS erfolgte in der Programmiersprache Java. Unter Berücksichtigung der einfachen Anbindung über die vorhandene, schlanke Schnittstelle des Constraint-Managers von YACS ist das Framework daher für Java-Anwendungen von besonderem Interesse. Die Entwicklung von YACS erfolgte, bedingt durch den Anwendungsfall EngCon unter einer Microsoft-Windows-Plattform auf x86-Hardware. Aufgrund der Plattformunabhängigkeit von Java sollte das YACS-Framework mit geringem Aufwand auch unter anderen Betriebssystemen und Rechnerarchitekturen lauffähig sein.

Der in diesem Kapitel abschließend erfolgte Vergleich mit weiterführenden Arbeiten zur Kombination und Kooperation von Constraint-Lösungsverfahren ermöglicht die Einordnung des YACS-Frameworks und einen Einblick darin, wie die Weiterentwicklung aussehen kann, z. B. hinsichtlich einer parallelen Ausführung unterschiedlicher Constraint-Solver innerhalb von YACS. Dies bedingt, dass der Fokus bei der Weiterentwicklung des Konzepts von YACS auf der flexiblen Steuerung von Kooperationen unterschiedlicher Solver sowie deren Informationsaustausch liegt.

[10] Bei Erreichen der Präzisionsgrenze werden, um sicherzustellen dass sich die exakte Lösung innerhalb des resultierenden Intervalls befindet, die Intervallgrenzen nach außen gerundet.

[11] Die Intervallarithmetik-Bibliothek IAMath ist die zurzeit einzige bekannte, in der Programmiersprache Java implementierte Lösung für Rechenoperationen mit reellwertigen Intervallen.

Kapitel 9

Zusammenfassung und Ausblick

Es gibt zwei Regeln für Erfolg im Leben:
1. Erzähle den Leuten nie alles, was du weißt...

Anonym

In diesem Kapitel erfolgt eine Zusammenfassung sowie ein kurzer Ausblick hinsichtlich der Erweiterungsmöglichkeiten der Konzeption und des entwickelten Prototypen.

9.1 Zusammenfassung

Das strukturbasierte Konfigurierungswerkzeug EngCon ist ein flexibles System, mit dem sich aufgrund einer austauschbaren Wissensbasis domänenunabhängig eine Vielzahl von Konfigurierungsproblemen behandeln lassen. Zur Unterstützung des Konfigurierungsvorgangs von EngCon ist eine Constraint-Komponente für die Verwaltung von algebraischen Abhängigkeiten vorgesehen. Aufgrund der Möglichkeit eine große Anzahl unterschiedlicher Konfigurierungsprobleme mit EngCon behandeln zu können, muss die Constraint-Komponente ebenfalls eine hohe Flexibilität aufweisen. Sie muss ferner die inkrementelle Verarbeitung von Constraint-Problemen erlauben, da durch die benutzergesteuerte, interaktive Konfigurierung das Constraint-Netz inkrementell anwächst.

Für den Konfigurierungsvorgang mit EngCon werden Lösungsmethoden für algebraische Constraint-Ausdrücke für finite und infinite Domänen benötigt. Finite Domänen werden durch diskrete Wertebereiche, infinite Domänen durch kontinuierliche, reellwertige Intervalle repräsentiert. Bestehende Constraint-Systeme erfüllen die Anforderungen für EngCon nur teilweise, zudem ist die Anbindung bestehender Systeme mit relativ hohem Integrationsaufwand verbunden. Die Eigenentwicklung geeigneter Constraint-Verfahren stellt daher eine angemessene Lösung dar. Eingebettet in eine modulare, objektorientierte Architektur ist für den Anwendungsfall EngCon ein hybrides Constraint-System erforderlich, welches Constraint-Solver für diskrete und kontinuierliche Werte in finiten und infiniten Domänen beinhaltet.

Die Effizienz von Constraint-Lösungsverfahren ist stark problemabhängig. Daher stellt es einen sinnvollen Ansatz dar, ein modulares Framework zu entwickeln, in das je nach Bedarf, und dem in der jeweiligen Wissensbasis definierten (Constraint-)Problem, unterschiedliche Constraint-Solver mit jeweils für das spezifische Problem effizienten Lösungsverfahren eingesetzt werden können. Ein objektorientiertes Constraint-Framework stellt die hierfür benötigten Rahmenbedingungen zur Verfügung, in denen inkrementelle Constraint-Lösungsverfahren implementiert werden können. Dies wird durch einheitliche Schnittstellen, einer klaren, objektorientierten Vererbungsstruktur innerhalb des Frameworks und einer Umgebung erreicht, welche die benötigten Komponenten für die Entwicklung und den Einsatz von Constraint-Solvern bereitstellt.

In dieser Arbeit wird ein Mechanismus entwickelt, mit dem sich unabhängig von der Implementierung jeweils geeignete Lösungsverfahren für unterschiedliche Problemstellungen definieren und anwenden lassen. Aufsetzend auf einer mehrschichtigen Architektur, dem YACS-Framework, ist der Wissensingenieur für EngCon in der Lage, flexibel und abstrahiert von den eigentlichen Lösungsalgorithmen zwischen unterschiedlichen Verfahren zum Auflösen von Constraint-Problemen mit unterschiedlichen Wertedomänen auszuwählen. Das YACS-Framework umfasst eine hierfür geeignete, hybride Schnittstelle basierend auf Constraint-Lösungsstrategien. Eingebettet in ein objektorientiertes Framework und gesteuert über Lösungsstrategien erhält der Benutzer Zugriff auf eine modulare Bibliothek von Constraint-Lösungsalgorithmen. Das YACS-Framework bietet somit eine flexible und erweiterbare Plattform für den problemabhängigen und anwendungsspezifischen Einsatz unterschiedlicher Constraint-Lösungsmethoden für finite und infinite Domänen.

Der in Java entwickelte Prototyp des YACS-Frameworks bietet bei dem derzeitigen Stand der Implementierung die folgenden Vorteile:

- YACS ist ein objektorientiertes Framework, wodurch die Flexibilität und Modularität, und damit die Austauschbarkeit der Lösungsmechanismen, und die Erweiterbarkeit um zusätzliche Lösungsverfahren sichergestellt wird.

- Die Schnittstelle sowie die implementierten Constraint-Solver ermöglichen einen inkrementellen Constraint-Lösungsprozess, der ein, bedingt durch den interaktiven Konfigurierungsprozess, inkrementell anwachsendes Constraint-Netz unterstützt.

- Durch eine phasenweise und durch separate, in XML definierte Constraint-Lösungsstrategien gesteuerte Verarbeitung der Constraints wird eine zusätzliche Abstraktionsebene eingeführt, die dem Wissensingenieur die einfache Anwendung unterschiedlicher Lösungsverfahren und Kombinationen dieser ermöglicht.

- Das YACS-Framework bietet eine hybride Schnittstelle und Unterstützung für finite und infinite Domänen. In der derzeitigen Implementierung existieren Constraint-Solver sowohl für diskrete als auch für intervallwertige Wertebereiche.

- Der Prototyp ermöglicht eine teilproblemübergreifende Metapropagation. Überschneidungen von Teilproblemen sind bisher allerdings ausschließlich bei identischen Domänen (finit/infinit) zulässig.

Der Einsatz des YACS-Frameworks ist nicht auf die Anwendung im Kontext von EngCon beschränkt. Aufgrund einer klaren und schlanken Schnittstelle ist YACS für unterschiedliche Anwendungen ohne großen Aufwand adaptierbar. Eine einfache, stringbasierte Schnittstelle zur Definition von Constraints und die Implementierung in der modernen Programmiersprache Java erleichtern die Nutzung von YACS als inkrementellen Inferenzmechanismus in Anwendungsszenarien, die eine Verarbeitung von intensional repräsentierten Constraints erfordern.

Mit der Realisierung des YACS-Frameworks wurde die Zielsetzung dieser Arbeit erreicht bzw. übertroffen. So lassen sich mit YACS nicht nur finite und infinite Domänen verarbeiten, sondern beliebig viele Constraint-Lösungsstrategien definieren und anwenden. Für identische Domänen (finit/infinit) ist zudem eine teilproblemübergreifende Metapropagation möglich. Die in dieser Arbeit vorgestellte Konzeption von YACS ist darüber hinaus tragfähig für weitergehende Möglichkeiten zur Constraint-Verarbeitung. Für Anwendungen, die nicht wie EngCon über einen übergeordneten Lösungsmechanismus verfügen, ist es wie in der Konzeption beschrieben möglich, in das YACS-Framework einen Meta-Constraint-Solver zu integrieren.

Der Prototyp von YACS geht bereits in seiner jetzigen Ausprägung über die Möglichkeiten von EngCon hinaus. Aufgrund konzeptioneller Beschränkungen von EngCon ist derzeit ausschließlich die *Propagation* der Constraint-Netze möglich, da die Schnittstelle zum Constraint-System von EngCon ausschließlich Wertebereichseinschränkungen durch Konsistenzverfahren zulässt und eine eigenständige Lösungssuche außerhalb von EngCon bisher nicht vorsieht. Vollständige Suchverfahren können in Bezug auf EngCon und ähnliche Anwendungsszenarien daher ausschließlich als Mechanismen zur Herstellung von globaler Konsistenz eingesetzt werden. Hierfür müsste eine Anpassung der jeweiligen Schnittstelle in der Form vorgenommen werden, dass die Ergebnisse des jeweiligen Suchvorgangs in die Domänen der Constraint-Variablen übertragen werden. Alternativ könnte dies innerhalb von YACS geschehen bzw. lassen sich vollständige Suchverfahren bereits innerhalb der jeweiligen Phase des Constraint-Lösungsprozesses wie Konsistenzverfahren zur Wertebereichseinschränkung nutzen.

Eine weitere Möglichkeit für den Einsatz von Suchverfahren bietet ein *look-ahead*-Mechanismus, mit dem sich während der Konfigurierung in definierten Abständen überprüfen ließe, ob für das Constraint-Problem noch eine Lösung existiert. Die Ermittlung sämtlicher Lösungen ist für dieses Anwendungsszenario nicht erforderlich, daher wäre eine einfache Suche innerhalb einer speziellen Constraint-Lösungsstrategie in diesem Fall ausreichend. Auch für einen derartigen *look-ahead*-Mechanismus sind allerdings konzeptionelle Erweiterungen von EngCon notwendig.

Das YACS-Framework wurde unter der *Lesser General Public License* (LGPL)[1] veröffentlicht und stellt somit Open-Source-Software dar. Neben der freien Verfügbarkeit und einer damit einhergehenden freien Verbreitungsmöglichkeit ist dies der Weiterentwicklung

[1] `http://www.gnu.org/licenses/lgpl.html`

des Frameworks förderlich. Durch Nutzung der LGPL betrifft dies auch die kommerzielle Verwendung und Weiterentwicklung des Frameworks. YACS wird als eigenständiges Projekt bei dem Open-Source-Dienstleister „SourceForge.net" gehostet, wo auch die Implementierung heruntergeladen werden kann.[2] Der Implementierungsstatus von YACS ist derzeit prototypisch. Neben intensiven Tests sind weitere Constraint-Lösungsmethoden wünschenswert. Den Vorzug sollten dabei weitere stabile Verfahren wie *Forward Checking* und *konfliktbasiertes Backjumping* erhalten. Für Intervalldomänen ist die Entwicklung von Preprozessing-Verfahren für die automatische Zerlegung komplexer Constraints in *solution functions* wünschenswert.

9.2 Ausblick

Wie bereits in Kapitel 8 angesprochen, stellt die Flexibilisierung der Kooperation, d. h. der Art und Weise, wie unterschiedliche Constraint-Solver miteinander interagieren, eine interessante Erweiterungsmöglichkeit für das YACS-Framework dar. Auch wenn ein grundlegendes Konzept von YACS der phasenweise Lösungsprozess ist, stellt eine parallele Verarbeitung von Constraint-Problemen eine erstrebenswerte Option dar.

Grundsätzlich bestehen sinnvolle Erweiterungen darin, weitere Mechanismen zur Unterstützung von dynamischen Aspekten entsprechend dem DCSP, CondCSP, CompCSP oder GCSP direkt in YACS zu integrieren. Wenn auch die implementierten Constraint-Lösungsverfahren grundsätzlich das inkrementelle Anwachsen des Constraint-Netzes unterstützen, fehlen einem reinen Constraint-Solver Automatismen wie Aktivitäts-Constraints, mit denen auf einer Metaebene definiert werden kann, zu welchem Zeitpunkt bestimmte Variablen und Constraints aktiviert werden sollen. Diese dynamischen Aspekte werden für gewöhnlich übergeordnet vom Constraint-System der Anwendung behandelt. Auch im Falle von EngCon existiert innerhalb des Konfigurierungswerkzeugs mit den konzeptuellen Constraints und dem Pattern-Matching-Mechanismus bereits ein dynamisches Constraint-System. Dennoch würde eine Integration zusätzlicher dynamischer Metaelemente in YACS eine Steigerung der Funktionalität bedeuten und das Framework attraktiver auch für Systeme machen, die bisher keine dynamischen Aspekte unterstützen.

Noch relevanter als das gesteuerte, dynamische Hinzufügen von neuen Variablen und Constraints könnte für spezielle Anwendungen das Entfernen bereits existierender Constraints durch Constraint-Relaxierung (bzw. *constraint retraction*) sein. Dies würde neben einem inkrementellen Anwachsen des Constraint-Netzes auch das effiziente, „dekrementelle" Zurücknehmen von Constraints erlauben, d. h. das Entfernen von Constraints, ohne das Constraint-Netz vollständig neu propagieren zu müssen.

Soll die Verarbeitung von überbestimmten Constraint-Problemen als PCSPs bzw. MaxCSPs, SoftCSPs, HCSPs oder Fuzzy CSPs möglich sein, so setzt dies jeweils Anpassungen am Konzept von YACS und der Konfigurierungs-Engine von EngCon voraus. Die benötigten Algorithmen zur Verarbeitung von Optimierungsproblemen, z. B. der Simplex-Algorithmus für reellwertige Domänen und *Branch & Bound* für finite Domänen, nutzen jeweils Zielfunktionen zur Generierung optimierter Lösungen. Für die hierfür benötigten

[2] `http://sourceforge.net/projects/constraints`

Informationen, z. B. hinsichtlich einer Zielfunktion oder einer speziellen Gewichtung, könnte im Framework eine allgemeine Oberklasse zur Repräsentation von Eigenschaften eines jeden Constraints vorgesehen werden.

Kommt ein Konfigurierungswerkzeug zum Einsatz, welches über die geschlossene Welt der *closed world assumption* (CWA) hinaus die *open-world*-Annahme ermöglicht, können die für das OCSP genannten Lösungsverfahren relevant werden. In einem solchen Szenario – aber auch für den allgemeinen Fall – können zudem asynchrone Mechanismen zum Auflösen von verteilten Constraint-Problemen (DisCSPs) angewendet werden. Eingebettet in z. B. ein verteiltes EJB-Framework müsste ein entsprechender Wrapper für YACS erstellt werden. Wobei sowohl von verteilten Konfigurierungs-Engines als auch von ausschließlich verteilten Constraint-Systemen ausgegangen werden kann.

Der Ansatz, ein flexibles Framework zu bieten, welches den Austausch und die Integration von Constraint-Lösungsverfahren vereinfacht, prädestiniert das YACS-Framework für ein adaptives Vorgehen in Bezug auf die Auswahl des jeweils für ein bestimmtes Problem einzusetzenden Lösungsverfahrens. Anhand der in dieser Arbeit benannten Kriterien und Eigenschaften für den effizienten Einsatz unterschiedlicher Constraint-Lösungsverfahren wird es dadurch möglich, innerhalb eines ACSPs den Prozess des Auswählens eines jeweils geeigneten Lösungverfahrens für ein konkretes (Teil-)Problem zu automatisieren.

Hinsichtlich der Präzision bei der Berechnung mit reellwertigen Intervallen ist festzuhalten, dass die Implementierung von YACS nicht grundsätzlich auf eine Intervallarithmetik-Bibliothek festgelegt ist. Wünschenswert wäre z. B. die (optionale) Unterstützung diskontinuierlicher Intervalle. Auch sollte sich der Präzisionsgrad, z. B. die Anzahl der Nachkommastellen, ab der zwei Intervalle als identisch betrachtet werden, von Fall zu Fall spezifizieren lassen. Außerdem sollten zukünftig weitere gängige mathematische Funktionen (z. B. *sin*, *cos*, *tan*, *log*, *ln*, *sqr*) unterstützt werden. Neben der eigentlichen Funktionalität, die auch in der zurzeit verwendeten Intervallarithmetik-Bibliothek IAMath bereits vorhanden ist, setzt dies eine Erweiterung der derzeitigen Implementierung des Constraint-Parsers voraus.

Neben zusätzlicher mathematischer Funktionalität bestehen Erweiterungsmöglichkeiten für das YACS-Framework in den bisher zulässigen Domänen. Für spezielle Anwendungen ist aus Effizienzgründen die native Unterstützung von booleschen Constraints denkbar. Auch die direkte Nutzung von reellen und rationalen Domänen, außer als spezieller Fall von reellwertigen Intervallen, kann, ebenso wie die Integration spezieller Algorithmen zu deren Auswertung, sinnvoll sein.

Aus Effizienzgründen könnte es ebenfalls notwendig werden, Teile des YACS-Frameworks (z. B. einzelne Constraint-Solver oder die Intervallarithmetik) oder das vollständige in dieser Arbeit vorgestellte Konzept in einer unter der jeweiligen Zielplattform kompilierbaren und nativ (ohne Java-Laufzeitumgebung) ausführbaren Programmiersprache zu reimplementieren.

Für weitergehende Tests, die Fortführung der Entwicklungstätigkeit, aber auch für die Anwender des Frameworks, würde eine Test-Konsole oder Constraint-Shell, und die damit verbundenen Möglichkeit, Constraint-Probleme komfortabel spezifizieren und einlesen zu können, eine signifikante Erleichterung darstellen. Das YACS-Framework könnte dadurch auf einfache Weise zu einer Test-Suite für unterschiedliche Constraint-Lösungsalgorithmen werden. Hierfür sind zudem zentrale Zeitnahme- sowie Zählmöglichkeiten (z. B. für die Anzahl der benötigten Konsistenztests) notwendig.

So wie im Rahmen der strukturbasierten Konfigurierung mit EngCon die Constraint-Verarbeitung genutzt wird, Konfigurationsabhängigkeiten zu repräsentieren und auszuwerten, existieren viele andere Anwendungen, in denen Constraints ebenfalls als Inferenzmechanismus dienen. Constraints werden aber nicht nur in Anwendungsprogrammen eingesetzt. Auch ein *Constraint-Programming*-System benötigt ein leistungsfähiges Constraint-System als Inferenzmechanismus und dient, wie z. B. EngCon oder eine mögliche Constraint-Shell, als Schnittstelle bzw. Anwendung für YACS. Eine interessante Aussicht stellt daher die Möglichkeit dar, das YACS-Framework als Basis für ein Java-basiertes CP-System zu nutzen. Unabhängig von der stringbasierten Schnittstelle von YACS würden sich die implementierten Framework-Klassen und deren Ausprägungen dazu nutzen lassen, ein derartiges System zu realisieren und eine Reihe von benötigten *global constraints* zu implementieren. Für die Verarbeitung generischer Constraints hingegen ließe sich die existierende, stringbasierte Schnittstelle von YACS einsetzen.

Teil IV

Anhänge

Anhang A

Konfigurierungswerkzeuge

In diesem Anhang befindet sich eine detaillierte Beschreibung der in Kapitel 2, Abschnitt 2.3 auf Seite 17 f. betrachteten Konfigurierungswerkzeuge. Sie stellen eine exemplarische Auswahl von Systemen dar, die z. T. in der Vergangenheit, auch für EngCon und dessen Vorläufer, eine wichtige Rolle für die Entwicklung wissensbasierter Methoden und Systeme gespielt haben. Zuerst werden einige „historische“ Systeme vorgestellt, die zu Beginn der Entwicklung in den 80er Jahren entstanden, danach folgt eine Reihe von Systemen, die zurzeit als kommerzielle Produkte erhältlich sind, sowie zwei Prototypen aus der aktuellen Forschung.

A.1 Historische Systeme

Ende der 80er Jahre stand die Softwareindustrie vor der sog. „Knowledge-Engineering-Lücke“. Entwickelte Expertensysteme wurden zum Großteil nicht eingesetzt: Der Zahl von mehreren Hundert in publizierten Übersichten aufgezählten Systeme für den deutschsprachigen Raum standen mehrere Dutzend tatsächlich eingesetzten Systeme gegenüber. Die Ursachen waren zumeist die mangelnde Portabilität, fehlende Integrationsfähigkeit, hoher Validierungs- und Wartungsaufwand und die Unterschätzung der Komplexität bei der Entwicklung solcher Systeme (vgl. Knemeyer und Schulenburg 1993; Simon 1993; Syska 1991; Weule 1993).

Hinzu kam, dass es so gut wie keine Entwurfshilfen für das „Knowledge-Engineering“ gab, wie sie für das Software-Engineering bereits zur Verfügung standen. Infolgedessen litten die entwickelten Systeme häufig unter mangelnder Zuverlässigkeit und Robustheit. Trotzdem etablierten sich eine Reihe von Systemen, die im Folgenden jeweils kurz angesprochen werden.

R1/XCON

Das Expertensystem XCon (ehemals R1) wurde zu Beginn der 80er Jahre entwickelt und von der Firma Digital Equipment Corporation (DEC) zur Konfigurierung von Kundenbestellungen verwendet. XCon ist ein regelbasiertes System, welches 1989 aus ca. 11.500 in der Sprache OPS5 (*Official Production System, Version 5*) spezifizierten Regeln bestand, die sich auf eine Datenbasis von ca. 30.000 Objekten bezogen (vgl. Günter 1992, S. 57; Neumann 1991, S. 15). Die Objekte werden in Komponentenbeschreibungen in einer Datenbank abgelegt und können von dort eingelesen werden. Das Wissen über erlaubte Teilkonfigurationen wird in Form von Regeln, sog. *production rules*, repräsentiert. Eine solche *production rule* besitzt einen linken Teil mit einer beliebigen Anzahl Bedingungen und einen rechten Teil mit einer beliebigen Anzahl Aktionen. Die Aufgabe wird in XCon nicht indirekt über funktionelle Anforderungen oder Beschränkungen spezifiziert, sondern konkret über eine Komponentenliste, welche die wesentlichen Bestandteile des Konfigurierungsziels enthält.

Ein großes Problem von XCon war die Gestaltung des Kontrollflusses der Konfigurierung, da Regeln nur eine schwache Struktur aufweisen (vgl. Günter 1992, S. 3). Sie sind daher i. A. nur für Domänen mit schwach ausgeprägten Kontrollstrukturen geeignet (vgl. Neumann 1991, S. 18). Um dem Abhilfe zu schaffen, wurden unterschiedliche Hilfsmittel zur Gestaltung des Kontrollflusses und später sogar eine eigens entwickelte Kontrollsprache (RIME) eingesetzt. Eine Eigenschaft dieser Bemühungen ist, dass die Abfolge der Konfigurierungsschritte in XCon in der Wissensbasis festgelegt ist und vom Benutzer zur Laufzeit nicht mehr geändert werden kann. XCon erlaubt aufgrund seiner Architektur keinerlei Interaktion mit dem Benutzer während des Konfigurierungsprozesses (vgl. Günter 1992, S. 49).

Neben den in Abschnitt 2.2.1 auf Seite 11 f. beschriebenen Problemen bei der Wissensakquisition aufgrund der Vermischung verschiedener Wissensarten, war der größte Nachteil dieses regelbasierten Konfigurierungswerkzeuges, dass jedes Jahr Software-Wartungen nötig wurden, die ca. die Hälfte aller Regeln betrafen. Die Pflege solcher Systeme ist somit als extrem aufwendig zu betrachten. Regelbasierte Systeme sind daher nur für überschaubare Domänen mit schwachen Kontrollstrukturen geeignet.

SICONFEX

Das Expertensystem Siconfex wurde für einen ähnlichen Anwendungsfall entwickelt wie XCon, bietet aber eine gänzlich andere Lösung an. Siconfex wurde 1985 u. a. bei Siemens, München, entwickelt. Mit Hilfe von Siconfex wurden die Betriebssysteme von SICOMP-Prozessrechnern konfiguriert (vgl. Neumann 1991, S. 18 f.). Die Eingabedaten bestehen aus Hardwarekomponenten und Angaben zur Software und der vorgesehenen Benutzungsart. Sie werden über einen benutzerfreundlichen Dialog abgefragt und erfordern nur noch ein Mindestmaß an Verständnis des Benutzers der zu konfigurierenden Hard- und Softwarekomponenten.

Neben der aufwendigen Benutzungsschnittstelle ist besonders die hochstrukturierte Wissensbasis kennzeichnend für Siconfex. Die Strukturierung des statischen Domänenwissens bezieht unterschiedliche Techniken ein: Objektbeschreibungen durch Schemata, konzeptuelle Hierarchien, Vererbungsmechanismen, Regeln, Domänenprozeduren und normaler Lisp-Code. Die Regelsemantik wird dabei im Gegensatz zu XCon nicht einfach prozedural als

```
IF <Bedingungen> THEN DO <Aktionen>
```

sondern logisch als

```
IF <Prämissen> THEN DO <Konklusionen>
```

interpretiert (vgl. Neumann 1988, S. 39). Neben einer hochstrukturierten Wissensbasis und einer umfangreichen Benutzungsschnittstelle verfügt SICONFEX über hybride Problemlösungstechniken und Optimierungsmethoden. Die Aufgabenspezifikation erfolgt in SICONFEX über einen algorithmischen Ansatz, wobei Regeln keinen Einfluss auf die Ablaufkontrolle haben.

MMC-Kon

Das System MMC-KON wurde Ende der 80er Jahre bei Siemens, Erlangen, zur Konfigurierung von Prozess-Steuerungssystemen als Prototyp entwickelt. Später wurde das System auf den Konzepten von PLAKON aufbauend, eines Vorläufers von ENGCON (vgl. Abschnitt 3.2, S. 20), reimplementiert. Der Anwendungsbereich von MMC-KON unterscheidet sich nicht wesentlich von den vorhergehenden, allerdings besitzt MMC-KON einige neue Funktionalitäten. So besteht das Domänenwissen aus konzeptuellen Beschreibungen der Objekte, die neben *is-a-* bereits über *has-parts*-Relationen innerhalb von Spezialisierungs- und Zerlegungshierarchien zueinander in Beziehung stehen (vgl. Abschnitt 2.2.2, S. 13). Unterschiedliche Sichten auf die Objekte erleichtern dabei die Wissensstrukturierung.

Die Reihenfolge der Konfigurierungsschritte lässt sich in MMC-KON ohne jegliche Kontrollflusskomponente vom Benutzer beinahe frei gestalten. Das System sorgt dafür, dass am Ende ein konsistentes Ergebnis vorliegt (vgl. Neumann 1991, S. 20 f.).

ALL-RISE

Das System ALL-RISE stammt aus einem anderen Anwendungsbereich als die bisher beschriebenen Systeme, nämlich dem Entwurf im Bauingenieurwesen. Auch wenn Entwurfssysteme meist nicht als Expertensysteme bezeichnet werden, besitzen sie einige Eigenschaften wie Wissensrepräsentationstechniken und Entwurfsmethoden, die für Expertensysteme interessant sind. Der Unterschied zu den bisher betrachteten Systemen besteht darin, dass der Schwerpunkt auf der Formgestaltung liegt und weniger auf dem Zusammenfügen von Komponenten. Formen besitzen viele Freiheitsgrade und sind daher schwieriger zu handhaben als Konfigurierungsprobleme, die auf Komponentenselektion beruhen (vgl. Neumann 1991, S. 22). Die Eingabe in ALL-RISE besteht aus einem räumlichen Plan eines Gebäudes in Form eines dreidimensionalen Gittermodells. Als Ausgabe wird eine Menge zulässiger „Struktursysteme“ geliefert, die jeweils bzgl. ihrer Eignung bewertet sind. Für den Entwurfsvorgang werden in einer Tiefensuche mittels Constraints aus einer Reihe von vordefinierten Struktursystemen, Teilsystemen und Komponenten die geeigneten herausgesucht.

Zur Repräsentation des Wissens in der statischen Wissensbasis werden die Schemata über die üblichen *is-a-* und *has-parts*-Relationen zueinander in Beziehung gesetzt (vgl. Abschnitt 2.2.2, S. 13). Eine besondere Bedeutung erfährt in diesem System bereits der

Wurzelknoten der Hierarchie. Er wird wie bei der strukturbasierten Konfigurierung als eine zulässige Konfiguration in ihrer abstraktesten Ausprägung interpretiert. Alle Nachfolger sind Spezialisierungen oder Teillösungen und dementsprechend Instanzen des Wurzelknotens. Zusätzlich zu den *is-a-* und *has-parts*-Relationen der Begriffshierarchie werden weitere Abhängigkeiten durch Constraints repräsentiert, durch die in der Wissensbasis definierte Komponenten zueinander in Beziehung stehen. Die Constraints werden instantiiert, propagiert und erfüllt, sobald die Ablaufkontrolle in der Wissenshierarchie auf sie stößt (vgl. Neumann 1988, S. 45 ff.).

Das All-Rise-System war für die Entwicklung von EngCon bzw. der Vorläufer von EngCon besonders wegen der Organisationsform der Wissensbasis und der intensiven Nutzung von Constraints interessant.

A.2 Verfügbare Produkte

Die folgenden Systeme wurden z. T. von Günter et al. (1999) hinsichtlich ihrer Eignung für die Konfigurierung variantenreicher Produkte untersucht. Der Fokus lag dabei auf der Funktionalität zur Konfigurierung von technischen Systemen. Typischerweise wird diese Tätigkeit von Vertriebsmitarbeitern zur Angebotserstellung und technischen Auftragsklärung durchgeführt. Dementsprechend sind fast alle Systeme (bis auf Cosmos) in der Lage, nach Beendigung der Konfigurierung ein Angebotsschreiben automatisch zu generieren. Alle Systeme benutzen zur Darstellung des Wissens eine Objektrepräsentation mit Parametern und nehmen während der Konfigurierung automatische Überprüfungen der Konsistenz (ob alle Abhängigkeiten erfüllt sind) und Vollständigkeit (ob alle notwendigen Komponenten enthalten sind) der Konfiguration vor. Außerdem erlauben alle Systeme die interaktive Konfigurierung durch den Anwender und bringen eine entsprechende (anpassbare) Benutzungsschnittstelle mit.

CAS-Konfigurator

Der CAS-Konfigurator der Firma SOLYP Informatik GmbH[1] ist ein Konfigurierungssystem zur Angebotserstellung. Die Wissensrepräsentation beruht auf einem hierarchischen Komponentenkatalog, der Vererbungsmechanismen, insbesondere auch Mehrfachvererbung zulässt. Außerdem stehen zur Wissensmodellierung Regeln und Entscheidungstabellen zur Verfügung. Die Modellierung des Wissens muss allerdings in der Programmiersprache C++ durchgeführt werden (vgl. Günter et al. 1999, S. 61 f.).

Während des interaktiven Konfigurierungsvorgangs, in dessen Verlauf der Benutzer Komponenten auswählt, kombiniert und parametrisiert, kann der Benutzer die Reihenfolge der Konfigurierungsschritte weitestgehend selbst beeinflussen. Entscheidungen die eindeutig möglich sind, werden automatisch durch das System getroffen. Auf Konfigurationskonflikte wird mit einem chronologischem Backtracking reagiert.

[1] `http://www.solyp.de`

Cameleon EPOS

Das von der Firma Access Commerce GmbH[2] vertriebene System CAMELEON EPOS (vormals ET-EPOS) beruht auf dem relativ einfachen Prinzip von Entscheidungsbäumen, die in einer speziellen Art von Tabellen realisiert werden. In den Tabellen werden die Abhängigkeiten der Konfiguration eingetragen und in einer vorgegebenen Reihenfolge abgearbeitet. Da die Kenntnis der Anwender über Tabellen relativ hoch ist, ist somit das Grundprinzip von CAMELEON EPOS in den meisten Fällen leicht verständlich, so dass auch die Wissensakquisition von „Nichtspezialisten“, d. h. Anwendern und Domänenexperten anstatt Informatikern oder Wissensingenieuren, durchgeführt werden kann (vgl. Günter et al. 1999, S. 62).

Leider versagt das System bei komplexen Anwendungen, in denen die Abläufe nicht fest vorgegeben werden können sondern heuristische Entscheidungen über komplexe Relationen notwendig sind. Trotz der Einfachheit (oder gerade deswegen) ist das System relativ stark auf dem Markt vertreten.

camos.Configurator

Der CAMOS.CONFIGURATOR (vormals SECON) wird von der Firma camos Software[3] vertrieben. Die Komponenten werden im CAMOS.CONFIGURATOR in einem „Klassenbaum“ repräsentiert. Neben einer Aggregationsbeziehung (*has-parts*) verfügt das System über Spezialisierungen (*is-a*), deren Eigenschaften während des Konfigurierungsverlaufs über Vererbungsmechanismen auf die „Nachfolger“ übertragen werden.

Die Pflege des System ist über eine separate Komponente möglich und kann vom Anwender selbst vorgenommen werden. Während der Konfigurierung kann der Anwender jederzeit erkennen ob Inkonsistenzen auftreten. Konflikte müssen manuell durch die Auswahl anderer, geeigneter Komponenten behoben werden. Heuristiken zur Automatisierung des Konfigurierungsvorgangs sind nicht vorgesehen.

Der CAMOS.CONFIGURATOR verfügt über mehrere Arten von Constraints: Zum einen lassen sich Konstrukte mittels der vorgegebenen Bausteine „kann“, „muss“, „darf nicht“ und „zuweisen“ (vorbelegen) erstellen, zum anderen ist die Möglichkeit gegeben, funktionale Zusammenhänge in Form von Prozeduren zu formulieren. Constraints werden im CAMOS.CONFIGURATOR bidirektional ausgewertet, sind aber nur passiv, d. h. sie beeinflussen nicht die Steuerung des Konfigurierungsverlaufs, was ihre Wirksamkeit nicht unerheblich einschränkt. Der Konfigurierungsvorgang erfolgt manuell, wird allerdings unterstützt durch die fortlaufende Konsistenzüberprüfung anhand der Constraints. Zur Steigerung der Akzeptanz bei den Benutzern des Systems können zudem Restriktionsverletzungen von Constraints ignoriert und zur späteren Auswertung in eine Liste zur „technischen Abklärung“ übernommen werden (vgl. Günter et al. 1999, S. 63).

[2] `http://www.access-commerce.de`
[3] `http://www.camos.de`

COMIX

Das System COMIX (COnfiguration of MIXingmachines) wurde am Fraunhofer-Institut für Informations- und Datenverarbeitung (IITB)[4] zur Konfigurierung von industriellen Rührwerken entwickelt (vgl. Sutschet 2001). COMIX ähnelt dem System PLAKON, ist jedoch vom Funktionsumfang her weniger mächtig. Zum Einsatz kommt ein Constraint-System, in dem „gerichtete“ Constraints verwendet werden. Durch diese Constraints wird das Kontrollwissen über die Domäne repräsentiert (vgl. Laudwein und Brinkop 1993). Aufgrund der speziellen Anforderungen der Anwendungsdomäne, wird eine vollständige Propagation durch ungerichtete Auswertung der Constraints nicht benötigt. Gerichtete Constraints berechnen Objekteigenschaften, sobald die erforderlichen Eingangsvariablen vorliegen. Es handelt sich hier daher im Grunde um vorwärtsverkettete Regeln (vgl. Abschnitt 2.2.1, S. 11).

COSMOS

Der Konfigurator COSMOS geht auf Verfahren und Algorithmen zurück, die 1993 von der DaimlerChrysler AG, Forschung und Technologie Berlin, entwickelt wurden (vgl. Günter et al. 1999, S. 62). Das System wird mittlerweile von der repas AEG Software GmbH[5] vertrieben. COSMOS basiert im Gegensatz zu den meisten anderen Tools und Forschungsprototypen auf dem Ansatz der ressourcenbasierten Konfigurierung (vgl. Abschnitt 2.2.4, S. 15).

COSMOS besitzt den Vorteil, dass die zur Verfügung stehenden, auf Ressourcen basierenden Wissensrepräsentationstechniken sehr anspruchsvoll sind und die Modellierung von komplexen Zusammenhängen ermöglichen. Allerdings muss die jeweilige Anwendung den Voraussetzungen zur ressourcenbasierten Konfigurierung genügen. d. h. es muss möglich sein, die Komponenten als Ressourcen zu modellieren und das System dementsprechend in Teilfunktionalitäten zerlegen zu können. Wenn dies der Fall ist, kann COSMOS an jeder Position in der Prozesskette (Entwicklung, Marketing/Vertrieb, Produktion) eingesetzt werden (vgl. Heinrich und Jüngst 1993).

KIKon

In dem Kooperationsprojekt „Kundenindividuelles Konfigurieren“ (KIKON) wurde durch die Deutsche Telekom AG[6], der Dialogis GmbH[7] und der GMD-Institute[8] für Autonome intelligente Systeme (AiS), für Angewandte Informationstechnik (FIT) und für Telekooperationstechnik (TKT) Techniken zur Konfigurierung von Telekommunikationsanlagen anwendungsorientiert umgesetzt.[9] KIKON ist ein ressourcenbasiertes System und basierte ursprünglich auf COSMOS. Weil die speziellen Anforderungen zur Konfigurierung von

[4]`http://www.iitb.fraunhofer.de`
[5]`http://www.repas.de`
[6]`http://www.telekom.de`
[7]`http://www.dialogis.de`
[8]`http://www.gmd.de`
[9]Die GMD ist im Jahr 2001 mit der Fraunhofer-Gesellschaft fusioniert. Die GMD-Einrichtungen wurden entsprechend in Fraunhofer-Institute umgewandelt (`http://www.fraunhofer.de`).

Telekommunikationsanlagen hiermit nur unzureichend umgesetzt werden konnten, wurde es vollständig neu entworfen und reimplementiert (vgl. Emde et al. 1996, S. 105).

In KIKon ist es möglich, fertige Konfigurationen zu „versiegeln“, und für die Wiederverwendung zu konservieren. Diese „versiegelten“ Konfigurationen sind als Komponenten zu betrachten, die als Teil der Wissensbasis somit das Produkt einer Wissensmodellierung sind. Gleichzeitig sind sie das Ergebnis eines Konfigurierungsprozesses. D. h. hier verwischt die Grenze zwischen dem Modellieren einer Wissensbasis und der eigentlichen Konfigurierung, da eine erstellte Konfiguration wiederum Teil eines weiteren Konfigurierungsvorgangs sein kann (vgl. Johnson-Schaaf 1999).

SAP Sales Configuration Engine

Die Sales Configuration Engine (SCE) von SAP[10] ist über eine Schnittstelle mit dem SAP R/3-Logistikmodul verbunden. Das System lässt sich „offline“ vom R/3-Gesamtsystem betreiben und verfügt im Vergleich zu dem vorhergehenden SAP R/3-Konfigurator über erweiterte Techniken zur Konfigurierung, die auch für technisch komplexere Systeme geeignet sind. Das Abhängigkeitswissen wird innerhalb von SCE mittels Regeln und Beschränkungen (*soft constraints*) repräsentiert. Als Problemlösungskomponente wird ein *Truth Maintenance System* (TMS) eingesetzt (vgl. Haag 1998). Auch wenn dadurch heuristische Konfigurierung unterstützt wird, bleibt ein großes Problem bestehen: Mit einem TMS lassen sich die Begründungen für Benutzerentscheidungen nicht erfassen (vgl. Günter et al. 1999, S. 65).

Baan SalesPlus

Der Konfigurator salesPLUS von Baan[11] wurde zur Unterstützung von kundenspezifischer Serienfertigung erstellt, d. h. die Produktkonfigurierung liefert kundenspezifisch angepasste Lösungen, die auf einem Standardprodukt bzw. -produktmodell basieren. Das Domänenwissen liegt in salesPLUS in einer „vorkompilierten“ Repräsentation vor, so dass die Wissensbasis sehr effizient verarbeitet werden kann. Das System ist constraintbasiert, und bietet unterschiedliche Constraint-Typen zur Unterstützung des Konfigurierungsprozesses. Spezielle *logic constraints* dienen dazu, die Kombination von Objekten zu kontrollieren. Mittels *arithmetic constraints* lassen sich physische Beschränkungen der Eigenschaften von Konfigurierungsobjekten definieren. Constraints, deren Verletzung erlaubt ist, werden *warning constraints* genannt. Eine Verletzung von *warning constraints* bedeutet, dass das zu konfigurierende System grundsätzlich konsistent ist, der Nutzer aber mit Einschränkungen zu rechnen hat

Die Konfigurierung in salesPLUS ist aufgrund der Vorgaben durch ein Standardprodukt nicht an eine bestimmte Reihenfolge der Konfigurierungsschritte oder eine Kontrollstrategie gebunden. Jederzeit lässt sich durch den Nutzer jede beliebige Konfigurierungsentscheidung treffen. Das Constraint-System sorgt dafür, dass die Konsistenz der Konfiguration gewahrt bleibt (vgl. Yu und Skovgaard 1998).

---

[10] `http://www.sap.de`
[11] `http://www.baan.com`

Tacton Configurator

Der Tacton Configurator wurde ursprünglich als ein wissenschaftlicher Prototyp namens Obelics (*OBjEct-oriented Language for ICS*) von 1992–1997 am *Swedish Institute of Computer Science* (SICS) entwickelt. Der Fokus von Obelics lag auf der Unterstützung interaktiver Konfigurierung (*Interactive Configuration by Selection, ICS*), wozu sowohl Techniken der logischen Programmierung als auch objektorientierte Verfahren in Form von SICStus Prolog und der objektorientierten Erweiterung SICStus Objects eingesetzt wurden (vgl. Axling und Haridi 1996).

Seit 1998 wird der Tacton Configurator als Produkt von der Firma Tacton Systems[12] vertrieben, einem Spin-Off des SICS. Der Tacton Configurator bietet sowohl ein dynamisches Constraint-System, welches ein während der Konfigurierung anwachsendes Constraint-Netz unterstützt, als auch eine sog. *Constructive Search Engine*. Diese generiert während des Konfigurierungsverlaufs Schritt für Schritt eine Lösung. Dabei werden nicht alle möglichen Lösungen berücksichtigt und bzgl. einer optimalen Lösung verglichen, sondern lediglich lokale Optimierungen vorgenommen. Wenn Lösungen für das Konfigurierungsproblem vorhanden sind, ist das Ergebnis jeweils eine einzige, geeignete Lösung, welche alle Anforderungen hinreichend erfüllt. Die Konfigurierung kann sowohl interaktiv als auch per Batch-Modus automatisch durchgeführt werden (vgl. Orsvärn und Axling 1999).

Lava

Das constraint-basierte Konfigurierungssystem Lava wurde im Auftrag der Siemens AG[13] zur Konfigurierung von komplexen Telekommunikationsanlagen erstellt. Lava konfiguriert Vermittlungsanlagen bestehend aus ca. 50.000 Komponenten (vgl. Fleischanderl 1999, S. 112). Das System nutzt im Kern eine Konfigurierungs-Engine namens Cocos, die das jeweilige Konfigurierungsproblem als *Generative Constraint Satisfaction Problem* abbildet und bearbeitet (vgl. Abschnitt 4.4.2, S. 58).

Die Repräsentation des Konfigurierungswissens geschieht in einer Wissensbasis, die aus hierarchischen Komponententyp-Beschreibungen und Constraints besteht. Die Wissensbasis wird in einer Sprache namens „ConTalk“ formuliert, deren Syntax an die Notation der objektorientierten Programmiersprache „Smalltalk“ angelehnt ist. Die Eigenschaften von Komponenten werden durch Attribute, Ports und Constraints definiert. Während Attribute z. B. numerische Eigenschaften einer Komponente beschreiben, werden durch Ports die Verbindungen zwischen unterschiedlichen Komponenten dargestellt. Wenn der Konfigurator auf eine Komponente stößt, deren Ports noch nicht belegt sind, werden den Ports entsprechende Komponenten ausfindig gemacht und eingetragen. Existieren diese noch nicht, werden passende neue Komponenten erzeugt. Das Constraint-System, welches die zwischen den Komponenten als Constraints repräsentierte Abhängigkeiten verwaltet, ist dementsprechend in der Lage, ein dynamisch anwachsendes Constraint-Netz zu verarbeiten, welches sich mittels generischer Constraints auf Komponententypen anstatt auf

[12] `http://www.tacton.com`
[13] `http://www.siemens.de`

konkrete Komponenteninstanzen bezieht. Lösungsstrategie des Constraint-Systems ist ein chronologischer Backtracking Algorithmus unterstützt durch Forward Checking (vgl. Abschnitt 5.2.4.5, S. 121).

Die Domäne, in der Lava zum Einsatz kommt, ist i. A. unterbestimmt und weist für ein Problem daher eine Vielzahl von Lösungen auf. Da es aus Komplexitätsgründen ineffizient ist, optimale Lösungen ausfindig zu machen, liefert das System, unterstützt durch entsprechende Variablen- und Werteordnungs-Heuristiken (vgl. Abschnitt 5.2.5, S. 128), konsistente Konfigurationen, die möglichst günstig sind und aus möglichst wenigen Komponenten bestehen. Dem zugute kommt die im System vorgesehene Trennung zwischen strategischem Wissen über die Konfigurierung und dem Wissen über die Konsistenz einer Konfiguration, so dass entsprechende Modifikationen unabhängig von den vorhandenen Constraints z. B. durch simples Verändern der Reihenfolge von Port-Variablen vorgenommen werden können (vgl. Fleischanderl et al. 1998).

ILOG (J)Configurator

Der ILOG Configurator bzw. ILOG JConfigurator (C++- und Java-Version) ist ebenfalls ein constraint-basiertes Konfigurierungswerkzeug (vgl. ILOG 2001, 2003). Ähnlich wie für Lava bzw. Cocos wurde von ILOG[14] das *Generative Constraint Satisfaction* zur Auflösung von komplexen Konfigurierungsproblemen durch ein sowohl generisches als auch dynamisches Constraint-System implementiert.

Der ILOG JConfigurator verfügt über eine Reihe von modernen (Web-)Schnittstellen, und kann relativ unproblematisch per EJB-Wrapper (*Enterprise Java Beans*) und *Web Services* in Java- oder C#-Applikationen integriert werden. Der ILOG Configurator und der ILOG JConfigurator sind Teil der ILOG Optimization Suite[15], in der sich zahlreiche constraint-basierte Werkzeuge zur Behandlung von (Zeit-)Planungs- und Konfigurierungsproblemen finden.

Die Inferenzmechanismen vom ILOG Configurator bzw. JConfigurator wurden zuletzt dahingehend erweitert, dass das Konfigurierungswissen in einer einfachen Art Beschreibungslogik repräsentiert wird. Constraints für eine konsistente Konfiguration werden durch diese logische Sprache spezifiziert, wodurch sie auf Objekte, die ebenfalls innerhalb dieser Beschreibungslogik vorliegen, angewendet werden können. Gleichfalls werden Konstrukte der Beschreibungslogik zur Klassifizierung in Constraints übersetzt, die wiederum mit den vorhanden Constraints interagieren (vgl. Junker und Mailharro 2003).

A.3 Forschungsprototypen

Im Folgenden werde zwei Prototypen aus der aktuellen Forschung vorgestellt. Beide setzten für den eigentlichen Konfigurierungsvorgang constraint-basierte Verfahren ein. ConBaCon behandelt den Spezialfall von *reconfiguration* mittels *bedingter Propagation* von Constraint-Netzen. CAWICOMS nutzt zwar constraint-basierte Verfahren zur Konfigu-

[14] http://www.ilog.com
[15] http://www.ilog.com/products/optimization/prods/index.cfm

rierung, der Schwerpunkt dieses Projekts liegt jedoch auf dem Aspekt der verteilten, webbasierten Konfigurierung.

ConBaCon

Die prototypische Implementierung von ConBaCon (*Constraint Based Configuration*) basiert auf der der CLP(FD)-Sprache CHIP (vgl. Abschnitt 4.5.1, S. 66). ConBaCon wurde am Institut für Rechnerarchitektur und Softwaretechnik (FIRST) der GMD[16] entwickelt und benutzt zur Spezifikation von Konfigurierungsproblemen eine eigens entwickelte Sprache namens ConBaConL. ConBaCon wurde bereits erfolgreich in der industriellen Steuerungstechnik eingesetzt, bspw. zur Konfigurierung von Stromversorgungsanlagen elektrischer Motoren.

Der Konfigurierungsprozess ist beeinflussbar durch Benutzerinteraktionen. Genau hier liegt der Schwerpunkt von ConBaCon: Im Bereich der Produktkonfigurierung kommt es immer wieder vor, dass einzelne Komponenten aufgrund von Kundenwünschen hinzugefügt oder ausgetauscht werden müssen (z. B. Upgrade). In diesem Fall wird nicht das gesamte Produkt ersetzt, sondern lediglich einzelne Komponenten. Dieser Vorgang wird *reconfiguration* genannt und kann globale Auswirkungen auf die gesamte Konfiguration nach sich ziehen (vgl. Sabin und Weigel 1998, S. 47 f.). Ziel von ConBaCon ist es, eine solche Rekonfigurierung möglichst so zu gestalten, dass von einer Änderung möglichst wenige Bereiche einer Konfiguration betroffen sind (vgl. John und Geske 1999b). Dazu wird u. a. eine *bedingte Propagation* mit Hilfe *weicher* Constraints (engl. *soft constraints*) eingesetzt, die sich von *harten* Constraints (engl. *hard constraints*) dadurch unterscheiden, dass sie nicht unbedingt erfüllt werden müssen, um zu einer konsistenten Konfiguration zu gelangen (vgl. John und Geske 1999a).

Aktuell wird versucht das System ConBaCon in Hinsicht auf die Behandlung umfangreicher Konfigurierungsprobleme zu verbessern. Dazu wird sowohl ein Clustering des Constraint-Netzes als auch ein Preprozessing zur Verringerung der Anzahl benötigter Constraints vorgenommen (vgl. John und Geske 2001, 2002).

CAWICOMS

Das Ziel des Projekts CAWICOMS[17] ist die Entwicklung eines B2B-Frameworks zur verteilten Produktkonfigurierung von Waren und Dienstleistungen. Der Prototyp, die *CAWICOMS Workbench*, bietet domänenabhängige Tools und Techniken, um eine einheitliche Basis zur Integration heterogener Konfigurierungssysteme zur Verfügung zu stellen. Mittels einfacher und offener Protokolle, zum verteilten Problemlösen und Austauschen komplexer Datenstrukturen, wird die Integration konfigurierbarer Unterprodukte von Lieferanten bzw. Zulieferern in einen Gesamtkontext ermöglicht. Der Benutzer bzw. der Kunde erhält den Eindruck, mit einem zentralen, homogenen System zu interagieren.

[16] `http://www.gmd.de`

[17] *Customer-Adaptive Web Interface for the Configuration of Products and Services with Multiple Suppliers*: `http://www.cawicoms.org`.

Die bisherigen Anwendungsszenarien von CAWICOMS stammen aus dem Bereich Telekommunikation (vgl. Ardissono et al. 2001, S. 2): Zum einen wurden die Funktionalitäten anhand der Konfigurierung, Orderung und Bereitstellung von Telefon-Vermittlungs-Systemen evaluiert. Dazu ist es erforderlich, entsprechende Module und Rechner physikalisch innerhalb einer Netzwerktopologie anzuordnen, zu verbinden und mit spezieller Software auszustatten. Zum anderen betrifft der zweite Anwendungsfall die Konfigurierung von Netzwerken und Services in der Domäne von IP-VPN (*Internet Protocol – Virtual Private Networks*). Auch dies umfasst sowohl technische Aspekte wie die Router-Konfigurierung und Bandbreiten-Reservierungen als auch Service-Dienstleistungen wie z. B. Installations-Support.

Zur Beschreibung einer allgemeinen Sprache, für die Repräsentation der Eigenschaften von konfigurierbaren Produkten und Dienstleistungen, setzt CAWICOMS auf eine ontologie-basierte Methode. In der Sprache DAML+OIL[18] wird innerhalb von CAWICOMS eine flexible Produkt-Ontologie für komplexe, anpassbare Produkte modelliert (vgl. Felfernig et al. 2002, S. 196 ff.). Der Ansatz ist eingebettet in das *Semantic-Web*-Konzept des World Wide Web Consortium (W3C)[19], dass sich in dieser Initiative darum bemüht, ein „semantisches" Verstehen von Inhalten im Internet voranzutreiben. Ziel ist es, Internet-basierte Technologien zu entwickeln, die es nicht nur Benutzern sondern auch autonomen Anwendungen ermöglichen, auf die Ressourcen des Internets zuzugreifen (vgl. Felfernig et al. 2002, S. 204).

Das konkrete Domänenwissen wird einheitlich auf auf XML-Basis (*eXtended Markup Language*) repräsentiert. Die Wissensakquisition der Produktmodelle wird UML-basiert (*Unified Modeling Language*) in grafischer Notation z. B. in kommerziellen Tools wie Rational Rose[20] vorgenommen und kann mittels OCL (*Object Constraint Language*) erweitert werden. Die CAWICOMS Workbench sieht Übersetzer für die vom Problemlösungsmechanismus benötigte, Java-basierte Wissensrepräsentation vor. In CAWICOMS wird ILOGs domänenunabhängiger, Java-basierter JConfigurator eingesetzt. JConfigurator implementiert *Generative Constraint Satisfaction* zur Auflösung von komplexen Konfigurierungsproblemen (vgl. Abschnitt 4.4.2, S. 58). Die eigentliche Problemlösung wird in einer verteilten Architektur durchgeführt, d. h. beteiligte Konfiguratoren haben nur jeweils eine Teilsicht auf das Produkt-Modell, wobei zwischen einem synchronen und einem asynchronen Konfigurierungsmodus unterschieden wird. Die Kommunikation zwischen den Konfiguratoren findet mittels XML-basiertem *SOAP messaging* (*Simple Object Access Protocol*) und *Web Services* statt. Die Komponenten der CAWICOMS Workbench selbst sind als *Enterprise Java Beans* (EJB) konform zu J2EE (*Java 2 Enterprise Edition*) implementiert (vgl. Ardissono et al. 2002, S. 620 f.).

Neben dem Aspekt der verteilten Konfigurierung wurde bei CAWICOMS besonderes Augenmerk auf die Benutzeroberfläche gelegt. Der Benutzer interagiert mit einer auf seine persönlichen Fähigkeiten bzgl. des Produktwissens angepassten Oberfläche. Aufgrund von unterschiedlichen Benutzerklassen generiert das System dynamische, web-basierte Benutzerinterfaces mittels JSP (*Java Server Pages*). Weiterhin fließen in die Generierung der

[18] `http://www.daml.org`
[19] `http://www.w3c.org`
[20] `http://www.rational.com`

Interfaces eine regelbasierte Personalisierung (ILOG *JRules*) und die Interpretation der Benutzerinteraktionen mittels Bayes'scher Netze ein (vgl. Ardissono et al. 2002, S. 621 f.).

Anhang B

Konfigurierungszyklus von EngCon

Im Folgenden wird der vollständige *zentrale Zyklus* des Kontrollmechanismus von EngCon in Ergänzung zu der verkürzten Darstellung in Kapitel 3, Abschnitt 3.5.2 auf Seite 31 ff. vorgestellt. Eine schematische Übersicht ist dem Flussdiagramm in Abbildung B.3 auf Seite 277 zu entnehmen. Die nachfolgende, sequentielle Abfolge des Zyklus orientiert sich an der Beschreibung von Ranze et al. (2002, S. 849 f.) und Günter (1992, S. 118 ff.):

1. *Auswahl der Aufgabenstellung*
   Auswahl eines von den in der Wissensbasis definierten Zielkonzepten. Das Zielkonzept definiert den Wurzelknoten der kompositionellen Hierarchie (siehe Abbildung 3.3, S. 25). Alle weiteren Konfigurierungsobjekte sind direkt oder indirekt Teil dieses Wurzelobjektes.

2. *Generieren der initialen Teilkonfiguration*
   Durch die Wahl der Aufgabenstellung wird die erste Teilkonfiguration erzeugt. Sie ist der Ausgangspunkt für alle weiteren Teilkonfigurationen und besteht bereits aus den benötigten Konfigurierungsobjekten. Diese Objekte sind Instanzen der Konzepte, die durch die Aufgabenstellung in der Begriffshierarchie spezifiziert sind.

3. *Bestimmen der aktuellen Strategie*
   Auswahl einer Strategie, die i. d. R. über mehrere Zyklen beibehalten wird. Der Konfigurierungsvorgang wird zur Strukturierung in unterschiedliche Phasen eingeteilt. Sie dienen dazu, einzelne Teilaufgaben zu bearbeiten, die relativ unabhängig voneinander sind. Phasen werden jeweils einer Strategie zugeordnet, in der das für die jeweilige Phase benötigte Kontrollwissen repräsentiert wird (vgl. Abschnitt 3.5.1, S. 30 und Abschnitt 3.5.2, S. 31). Die Auswahl einer Strategie erfolgt automatisch anhand der höchsten Priorität der verfügbaren Strategien. Der Benutzer kann sie allerdings auch manuell auswählen. Ein „Experte“ erhält somit die Möglichkeit, die Konfigurierung weitestgehend selbst zu beeinflussen.

   Wenn alle Strategien abgearbeitet wurden, wird die sog. „Default-Strategie“ aufgerufen. Sie besitzt die geringste Priorität und listet alle verbleibenden Konfigurie-

```
(def-ks-fokus
    :name   peripherie-konfigurierung-fokus
    :slot   ((:nicht Preis))
    :do     ((Monitor)
             (Tastatur)
             (Maus)
             (Drucker)
             (Scanner)
             (Boxen_Set)
             (Joystick))
    :ks-typ ())
```

Abbildung B.1: Konfigurierungsschritt-Fokus (KS-Fokus)

rungsschritte in der Agenda auf, die vom Wissensingenieur nicht explizit aufgeführt wurden.

4. *Generieren der Agenda*
Die Reihenfolge der Konfigurierungsschritte ist entscheidend für eine erfolgreiche Konfigurierung. Der Kontrollmechanismus überprüft dazu, welche Konfigurierungsschritte jeweils noch möglich sind und fasst diese Schritte in einer Agenda zusammen. Um mögliche Konfigurierungsschritte ausfindig zu machen, erfolgt ein Vergleich der Konfigurierungsinstanzen mit den zugehörigen Konzepten der Begriffshierarchie. Für Instanzen, die noch nicht vollständig spezialisiert sind, werden die entsprechenden Agenda-Einträge generiert.

   Die in der jeweiligen Strategie definierten KS-Foki werden zur Optimierung eingesetzt (vgl. Abschnitt 3.5.1, S. 30). Ein KS-Fokus wird dazu genutzt, die Agenda auf einen bestimmten Teil der aktuellen Teilkonfiguration einzuschränken bzw. zu fokussieren (siehe Abbildung B.1). Die Reihenfolge der Konfigurierungsschritte in der Agenda leitet sich her aus den in der aktuellen Strategie priorisierten Agenda-Auswahlkriterien. Die Priorität ist durch die dort definierte Reihenfolge der Auswahlkriterien repräsentiert.

   Weil sich die Erzeugung der Agenda an der Begriffshierarchie orientiert, in der in einem Metamodell entsprechend der CWA die generische Beschreibung aller vollständigen, zulässigen Konfigurierungen definiert ist, wird gewährleistet, dass in der Agenda für alle notwendigen Konfigurierungsschritte stets auch die erforderlichen Agenda-Einträge existieren. Wenn in der Agenda keine Einträge mehr vorhanden sind, ist die Konfigurierung entweder vollständig spezifiziert, oder es sind alle Konfigurierungsschritte der aktuellen Phase (Strategie) durchgeführt worden. Als nächstes würde dementsprechend versucht werden, eine neue Strategie auszuwählen (Schritt 3).

5. *Auswahl aus der Agenda*
Mit Hilfe von Agenda-Auswahlkriterien wird ein Konfigurierungsschritt zur Durch-

```
(def-auswahlkriterium
    :name   bevorzuge-pc-zerlege-peripherie
    :slot   ((hat-peripherie))
    :do     ((PC))
    :ks-typ ((zerlege)))
```

Abbildung B.2: Agenda-Auswahlkriterium

führung aus der erzeugten Agenda ausgewählt. Ein Agenda-Auswahlkriterium ist ein Muster, das durch die Angabe einer teilweisen Spezifikation eines Konfigurierungsschrittes repräsentiert wird.[1] Es kann sich auf Konfigurierungsobjekte (`do`), Eigenschaften von Konfigurierungsobjekten (`slot`) oder KS-Typen (`ks-typ`) beziehen (siehe Abbildung B.2). Die Agenda-Einträge werden mit dem Muster verglichen. Ein erfolgreicher Vergleich bedeutet, dass der Eintrag das Kriterium erfüllt.

Die Auswahlkriterien sind innerhalb der Strategie in einer strikten Ordnung definiert. Die Reihenfolge steht für die Priorität der Kriterien. Liefert die Anwendung eines Auswahlkriteriums mehr als einen Konfigurierungsschritt, werden der Reihe nach die folgenden Kriterien angewendet, bis nur genau ein Schritt übrig bleibt.[2]

6. *Auswahl des Bearbeitungsverfahrens*
   Ein Bearbeitungsverfahren ist eine Methode zur Durchführung eines Konfigurierungsschrittes. Die möglichen Verfahren sind unter Schritt 7 aufgeführt. Die Auswahl des jeweiligen Bearbeitungsverfahrens orientiert sich wiederum an der Reihenfolge, in der die Verfahren in der Strategie spezifiziert sind. Kann ein Verfahren nicht angewandt werden (z. B. wenn kein Default-Wert vorhanden ist), wird das jeweils nächste Bearbeitungsverfahren verwendet. Sind keine automatischen Verfahren anwendbar, wird eine Eingabe des Benutzers erwartet.

7. *Anwendung des Bearbeitungsverfahrens*
   Durch die Anwendung eines Bearbeitungsverfahrens wird ein Konfigurierungsschritt durchgeführt. Hierbei wird für den betreffenden Konfigurierungsschritt ein Wert bzw. ein Ergebnis ermittelt oder ein Wertebereich eingeschränkt. Folgende Bearbeitungsverfahren sind vorgesehen:[3]

[1] In PlaKon und KonWerk waren noch eine Reihe weiterer Auswahlkriterien vorgesehen, auch auf Planungsaufgaben bezogen, die jedoch von EngCon nicht implementiert werden.

[2] Bei einem nicht eindeutigen Ergebnis wird aus den verbleibenden Konfigurierungsschritten eine zufällige Auswahl getroffen.

[3] In PlaKon war als weiteres Bearbeitungsverfahren die Propagation des Constraint-Netzes vorgesehen. Es gab in diesem System die Möglichkeit, dass Constraint-Netz aus Effizienzgründen nicht bei jedem Durchlauf des Zyklus zu propagieren. In KonWerk hatte man von diesem Vorgehen allerdings bereits Abstand genommen, da es generell wünschenswert ist, möglichst schnell konsistente Werte im Constraint-Netz zu erhalten, und der verwendete Constraint-Solver entsprechend leistungsfähiger war.

- *Default*: Übernahme eines statischen Wertes (z. B. 2.0 GHz für die Taktung eines Prozessors).
- *dynamischer Default*: Ein dynamischer Default ist eine Funktion zur Auswahl aus dem zulässigen Wertebereich eines Slots (z. B. Mittelwert oder Minimum).
- *Berechnungsfunktion*: Eine beliebige durch den Wissensingenieur definierbare Funktion, die zur Auswertung des Bearbeitungsverfahrens aufgerufen wird. Die Berechnungsfunktion wird nur einmalig zur Durchführung des Konfigurierungsschrittes durchgeführt.
- *allgemein zulässiger Wertebereich* (AZW): Mit diesem Bearbeitungsverfahren wird verhindert, dass dem Benutzer jeder z. B. durch ein Constraint automatisch festgelegte Slot-Wert zur Bestätigung bzw. Änderung anschließend nochmals vorgelegt wird, wie es das Standardverfahren wäre. Die Änderung eines solchen Wertes durch den Benutzer würde zu einer Inkonsistenz führen und einen Konflikt auslösen, der entsprechend behoben werden müsste (siehe unten).
- *Benutzereingabe*: Es wird eine Anfrage an den Benutzer vorgenommen, um den Wert für einen Konfigurierungsschritt zu ermitteln. Der Benutzer besitzt dabei die Möglichkeit, den aktuellen Konfigurierungsschritt zurückzustellen oder andere Bearbeitungsverfahren einzusetzen.

Die möglichen Bearbeitungsverfahren lassen sich unproblematisch erweitern. Über eine allgemeine Schnittstelle können beliebige Verfahren zur Berechnung eines Konfigurierungsschrittes eingebunden werden. Denkbar sind z. B. Bibliotheks- bzw. fallbasierte Lösungen, die Auswertung von Simulationsprogrammen, etc. (vgl. Günter 1991a, S. 101).

8. *Berechnung der Auswirkungen eines Konfigurierungsschrittes*
Nach der Ausführung des Konfigurierungsschrittes, müssen mögliche Auswirkungen auf andere Konfigurierungsobjekte berücksichtigt werden. Dies können taxonomische Inferenzen und Auswirkungen das Constraint-Netz betreffend sein, die entsprechend berechnet werden müssen. Taxonomische Inferenzen bewirken die Spezialisierung von Komponenten bzw. Aggregaten (vgl. Abschnitt 3.4.4, S. 27). Die Constraint-Propagation dient der Validierung, ob die aktuelle Teilkonfiguration weiter eingeschränkt werden kann bzw. ob sich Inkonsistenzen ergeben haben (vgl. Abschnitt 3.6, S. 33).[4] Weiterhin prüft der Pattern-Matcher, ob neue Constraints zum Constraint-Netz hinzugefügt werden müssen, weil z. B. neue Komponenten in der aktuellen Teilkonfiguration erzeugt worden sind (vgl. Abschnitt 3.6.1, S. 35). Taxonomische Inferenzen und Constraints werden so lange propagiert, bis sich keine Änderungen mehr ergeben.

9. *Auflösen von Konflikten*
Stellen sich Konfigurierungsentscheidungen als falsch heraus, führt dies zu Konflikten

[4] PlaKon unterstützte die Möglichkeit, aus Effizienzgründen lediglich Teile des Constraint-Netzes zu propagieren. EngCon hingegen propagiert immer das vollständige Netz, um zu gewährleisten, dass alle Abhängigkeiten bei jedem Zyklus erfasst werden.

bzw. Inkonsistenzen in der aktuellen Teilkonfiguration. Ein inkonsistenter Zustand ergibt sich z. B. dann, wenn sich ein Oberkonzept aufgrund eines Konflikts mit einem anderen Konzept nicht zu einem seiner Unterkonzepte spezialisieren lässt. Die Wissensbasis definiert in diesem Fall nicht das für den aktuellen Konfigurierungszustand benötigte Konzept. EngCon unterstützt derzeit die Anwendung eines Konfliktauflösungsverfahrens in Form eines einfachen „Undo“ durch den Benutzer.[5]

10. *Aktualisieren der Agenda*
Zum einen werden die Einträge in der Agenda überprüft, ob sie aufgrund von Einschränkungen durch Constraints oder taxonomischen Inferenzen terminal sind und entsprechend entfernt werden können. Zum anderen wird geprüft, ob neue Einträge zur Agenda hinzugefügt werden müssen. Wenn die Agenda leer sein sollte, wird mit Schritt 3 die nächste Strategie ausgewählt, ansonsten fährt der Kontrollmechanismus fort mit Schritt 5, der Auswahl des nächsten Konfigurierungsschrittes aus der Agenda.

[5] Im Rahmen von EngCon wurde weiterhin eine Form von abhängigkeitsgesteuertem Backtracking untersucht, bei dem, gestützt auf ein Abhängigkeitsnetz, versucht wird zu ermitteln, welche Konfigurierungsentscheidung den Konflikt verursacht hat. Der Benutzer wird auf diese Weise zu der Stelle geführt, wo der Konflikt ausgelöst wurde. Diese Konfliktbehandlung wurde bisher allerdings lediglich prototypisch umgesetzt.

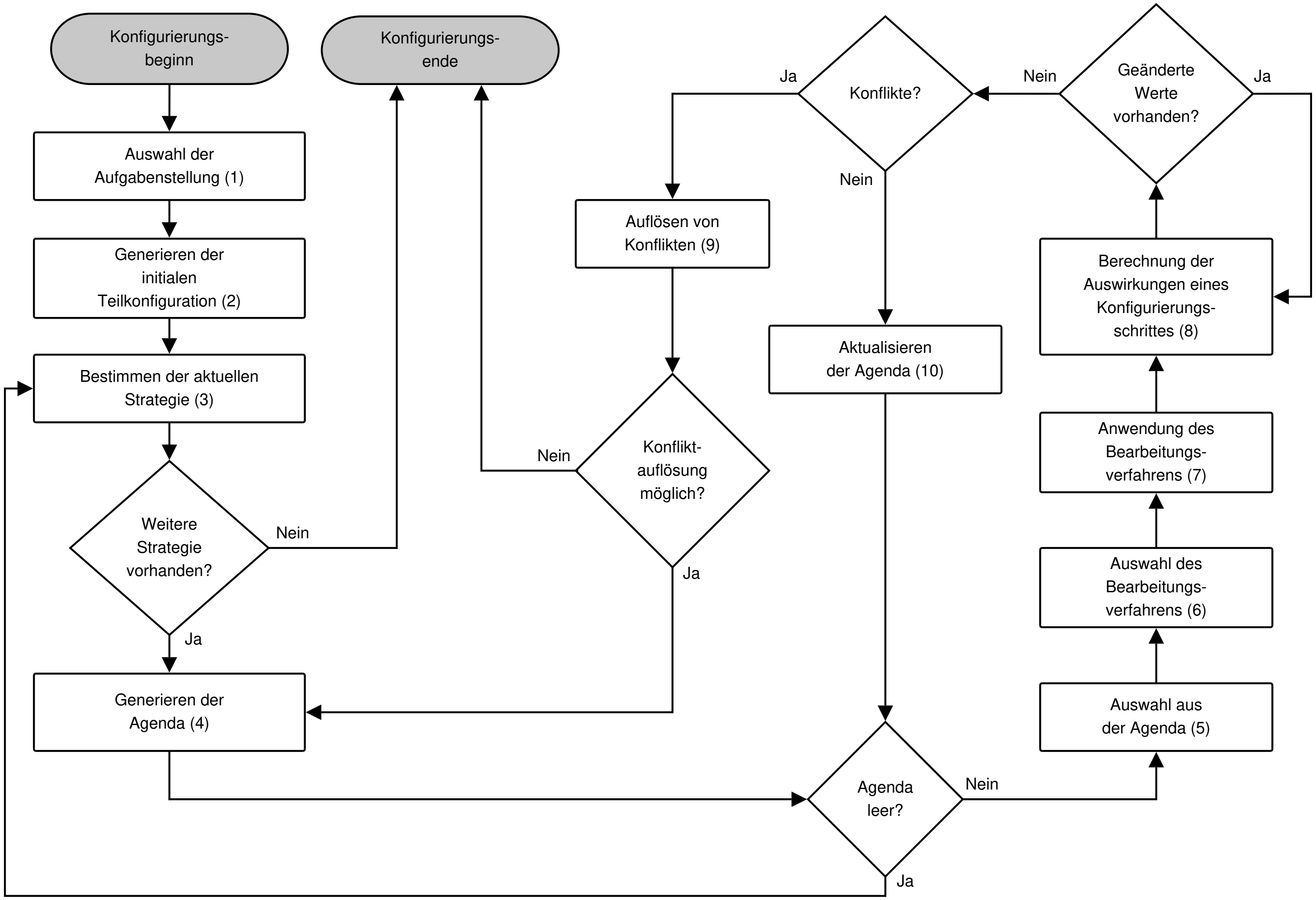


Abbildung B.3: Der vollständige *zentrale Kontrollzyklus* (vgl. Ranze et al. 2002, S. 849 f.)

Anhang C

Constraint-Lösungsstrategien

An dieser Stelle sind entsprechend der Beschreibung in Kapitel 7, Abschnitt 7.7 auf Seite 213 ff. die XML-DTD von YACS sowie einige einfache XML-Beispielstrategien spezifiziert (vgl. Eckstein 2000). Durch die hier aufgeführten Lösungsstrategien werden außerdem während der Validierung in Kapitel 8, Abschnitt 8.3 auf Seite 225 ff. die dort genannten Problemstellungen verarbeitet.

Zuerst folgt der Inhalt der Datei `yacs_strategies.dtd`, in der die XML-DTD für die vom YACS-Framework interpretierbaren Constraint-Lösungsstrategien definiert ist:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Simple XML-DTD for strategies of the YACS framework. -->

<!ELEMENT strategies (strategy+)>
<!ELEMENT strategy (preprocessing, consistency, search)>
<!ATTLIST strategy name ID #REQUIRED>
<!ELEMENT preprocessing (solver*)>
<!ELEMENT consistency (solver*)>
<!ELEMENT search (solver*)>
<!ELEMENT solver (#PCDATA)>
```

Entsprechend dieser DTD enthalten Lösungsstrategien neben ihrem Namen als Parameter drei Abschnitte, welche die jeweilige Phase des Constraint-Lösungsprozesses repräsentieren: `preprocessing`, `consistency` und `search`. In jedem dieser Abschnitte lassen sich jeweils innerhalb von `solver`-Tags eine Reihe von Constraint-Solvern angeben, die durch ihren vollständigen Klassennamen inkl. des Package-Pfades spezifiziert werden müssen.

Es folgt abschließend der Inhalt der Datei `yacs_strategies.xml`, in der eine Reihe von einfachen Beispielstrategien definiert sind:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE strategies SYSTEM "yacs_strategies.dtd">

<strategies>

  <strategy name="low_consistency">
    <preprocessing>
```

```
    </preprocessing>
    <consistency>
      <solver>
        yacs.solver.fdsolver.consistency.NCSolver
      </solver>
    </consistency>
    <search>
    </search>
  </strategy>

  <strategy name="medium_consistency">
    <preprocessing>
    </preprocessing>
    <consistency>
      <solver>
        yacs.solver.fdsolver.consistency.NCSolver
      </solver>
      <solver>
        yacs.solver.fdsolver.consistency.AC3Solver
      </solver>
    </consistency>
    <search>
    </search>
  </strategy>

  <strategy name="simple_search">
    <preprocessing>
    </preprocessing>
    <consistency>
    </consistency>
    <search>
      <solver>
        yacs.solver.fdsolver.search.SingleSolutionBTSolver
      </solver>
    </search>
  </strategy>

  <strategy name="search">
    <preprocessing>
    </preprocessing>
    <consistency>
    </consistency>
    <search>
      <solver>
        yacs.solver.fdsolver.search.BacktrackingSolver
      </solver>
    </search>
  </strategy>

  <strategy name="search_with_look-ahead">
    <preprocessing>
    </preprocessing>
    <consistency>
    </consistency>
```

```
    <search>
      <solver>
        yacs.solver.fdsolver.search.MAC3Solver
      </solver>
    </search>
  </strategy>

  <strategy name="interval_consistency">
    <preprocessing>
    </preprocessing>
    <consistency>
      <solver>
        yacs.solver.intervalsolver.consistency.HullConsistencySolver
      </solver>
    </consistency>
    <search>
    </search>
  </strategy>

</strategies>
```

Anhang D

Grammatik der Constraint-Ausdrücke

In diesem Anhang ist die Parser-Grammatik der von YACS per JLex/Java CUP interpretierbaren Constraint-Ausdrücke dokumentiert (vgl. Abschnitt 7.5, S. 204 ff.). Zuerst wird die vom Parser zum Einlesen von gültigen Zeichen benötigte Datei `scanner.lex` für die lexikalische Analyse durch JLex aufgeführt (vgl. Berk 2000). Sie befindet sich im Package `yacs.parser`:

```
package yacs.parser;

import java_cup.runtime.Symbol;

%%

%cup
%public

ALPHA=[A-Za-z]
DIGIT=[0-9]

%%

";"                        {return new Symbol(sym.SEMI);}
","                        {return new Symbol(sym.COMMA);}
"="                        {return new Symbol(sym.EQUAL);}
"!="                       {return new Symbol(sym.NOT_EQUAL);}
">"                        {return new Symbol(sym.GREATER);}
"<"                        {return new Symbol(sym.LOWER);}
">="                       {return new Symbol(sym.GREATER_EQUAL);}
"<="                       {return new Symbol(sym.LOWER_EQUAL);}
"+"                        {return new Symbol(sym.PLUS);}
"-"                        {return new Symbol(sym.MINUS);}
"*"                        {return new Symbol(sym.TIMES);}
"/"                        {return new Symbol(sym.DIVIDE);}
"("                        {return new Symbol(sym.LPAREN);}
")"                        {return new Symbol(sym.RPAREN);}
"["                        {return new Symbol(sym.LBRACKET);}
```

```
"]"                              {return new Symbol(sym.RBRACKET);}
{DIGIT}+                         {return new Symbol(sym.NUMBER,
                                              new Integer(yytext()));}
{DIGIT}+"."{DIGIT}+              {return new Symbol(sym.FLOAT,
                                              new Double(yytext()));}
{ALPHA}({ALPHA}|{DIGIT}|_|".")*  {return new Symbol(sym.VARIABLE, yytext());}
[ \t\r\n\f\b]                    {/* ignore white space. */}
.                                {System.err.println("Illegal character: "
                                               +yytext());}
```

Von Scanner erfasste Zeichen werden an den Parser weitergegeben und anhand der Java-CUP-Grammatik in der Datei `parser.cup` ausgewertet (vgl. Hudson 1999). Diese Datei befindet sich ebenfalls im Package `yacs.parser` und hat den folgenden Inhalt:

```
package yacs.parser;

import java.util.HashMap;

import yacs.domain.*;

import java_cup.runtime.*;

action code {:
  /* this is where the action code goes */

  /** der erzeugte Ausdruck */
  public Expression expression;

  /** HashtMap zum "Wiederfinden" der generierten Variablen */
  public HashMap actionVariablesMap = new HashMap();

  /** temporaere Variable zum Zwischenspeichern */
  public Variable tmpVariable;

:};

parser code {:
 /* this is where the parser code goes */

  /** HashtMap zum Zwischenspeichern der bereits vorhandenen Variablen */
  public HashMap parserVariablesMap = new HashMap();

 /**
  * Hinzufuegen bereits vorhandener Variablen. Anstatt neue Variablen
  * zu erzeugen, werden so die Referenzen bestehender Objekte
  * verwendet.
  * @param variablesMap HashMap
  */
  public void addVariablesMap(HashMap variablesMap) {
    this.parserVariablesMap.putAll(variablesMap);
  }

 /**
  * Gibt den gescannten und geparsten Ausdruck als Expression zurueck.
```

```
   * @return Expression
   */
  public Expression expression() {
    return action_obj.expression;
  }

:};

init with {:

  // Die in "parser" bereits befindlichen Variablenreferenzen in das
  // "action_obj" uebertragen. Dies muss hier separat geschehen, da
  // erst waehrend der "init"-Phase das "action_obj" instantiiert ist.
  action_obj.actionVariablesMap.putAll(this.parserVariablesMap);

:};

/* Terminals (tokens returned by the scanner). */
terminal           UMINUS, SEMI, LPAREN, RPAREN, COMMA, LBRACKET, RBRACKET;
terminal           PLUS, MINUS, TIMES, DIVIDE;
terminal           EQUAL, NOT_EQUAL, GREATER, LOWER, GREATER_EQUAL, LOWER_EQUAL;
terminal Integer   NUMBER;
terminal Double    FLOAT;
terminal String    VARIABLE;

/* Non Terminals */
non terminal              expr_all;
non terminal Expression   expr_list, expr, expr_part;

/* Precedences */
precedence left EQUAL, NOT_EQUAL, GREATER, LOWER, GREATER_EQUAL, LOWER_EQUAL;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left UMINUS;

/* The grammar */

expr_all    ::= expr_list:e
                {:
                 System.out.println(" => "+e.toString());
                 this.expression = e;
                :}
              ;

expr_list   ::= expr:e
                {: RESULT = e; :}
              | expr_list:l expr:r
                {: RESULT = new BinaryOperator(sym.SEMI, ";", l, r); :}
              ;

expr        ::= expr_part:l EQUAL expr_part:r SEMI
                {: RESULT = new BinaryOperator(sym.EQUAL, "=", l, r); :}
              | expr_part:l NOT_EQUAL expr_part:r SEMI
                {: RESULT = new BinaryOperator(sym.NOT_EQUAL, "!=", l, r); :}
```

```
                | expr_part:l GREATER expr_part:r SEMI
                  {: RESULT = new BinaryOperator(sym.GREATER, ”>”, l, r); :}
                | expr_part:l LOWER expr_part:r SEMI
                  {: RESULT = new BinaryOperator(sym.LOWER, ”<”, l, r); :}
                | expr_part:l GREATER_EQUAL expr_part:r SEMI
                  {: RESULT = new BinaryOperator(sym.GREATER_EQUAL, ”>=”, l, r); :}
                | expr_part:l LOWER_EQUAL expr_part:r SEMI
                  {: RESULT = new BinaryOperator(sym.LOWER_EQUAL, ”<=”, l, r); :}
                ;

expr_part   ::= NUMBER:n
                  {: RESULT = new Constant(n); :}
                | LBRACKET FLOAT:lo COMMA FLOAT:hi RBRACKET
                  {: RESULT = new Constant(lo, hi); :}
                | LBRACKET NUMBER:lo COMMA NUMBER:hi RBRACKET
                  {: RESULT = new Constant(lo.doubleValue(), hi.doubleValue()); :}
                | LBRACKET FLOAT:lo COMMA NUMBER:hi RBRACKET
                  {: RESULT = new Constant(lo.doubleValue(), hi.doubleValue()); :}
                | LBRACKET NUMBER:lo COMMA FLOAT:hi RBRACKET
                  {: RESULT = new Constant(lo.doubleValue(), hi.doubleValue()); :}
                | VARIABLE:v
                  {: tmpVariable = (Variable)actionVariablesMap.get(v);
                     if (tmpVariable == null) {
                       // Variable mit leerer Domaene instantiieren:
                       tmpVariable = new Variable(v, new Domain());
                       actionVariablesMap.put(v, tmpVariable);
                     }
                     RESULT = tmpVariable;
                  :}
                | expr_part:l PLUS expr_part:r
                  {: RESULT = new BinaryOperator(sym.PLUS, ”+”, l, r); :}
                | expr_part:l MINUS expr_part:r
                  {: RESULT = new BinaryOperator(sym.MINUS, ”−”, l, r); :}
                | expr_part:l TIMES expr_part:r
                  {: RESULT = new BinaryOperator(sym.TIMES, ”*”, l, r); :}
                | expr_part:l DIVIDE expr_part:r
                  {: RESULT = new BinaryOperator(sym.DIVIDE, ”/”, l, r); :}
                | MINUS expr_part:e
                  {: RESULT = new UnaryOperator(sym.UMINUS, ”−”, e); :}
                  %prec UMINUS
                | LPAREN expr_part:e RPAREN
                  {: RESULT = e; :}
                ;
```

Anhang E

YACS API-Dokumentation

Nachfolgend ist die API-Dokumentation des in Kapitel 7 auf Seite 199 ff. beschriebenen YACS-Frameworks aufgeführt. Attribute, Standardkonstruktoren und *private*-Methoden werden nicht aufgelistet, ebenso werden, um diese Übersicht kompakt zu halten, geerbte Methoden von übergeordneten Klassen nicht separat ausgewiesen (außer sie werden durch eine spezielle Funktionalität überschrieben). Abstrakte Methoden und Implementierungen von Methoden aus Interfaces werden in den Unterklassen ebenfalls nicht nochmals aufgeführt. Deren Dokumentation kann den jeweils übergeordneten Klassen entnommen werden.

E.1 Package `yacs`

Enthält das Interface für den YACS Constraint-Manager bzw. die Implementierung dessen.

E.1.1 Interface `YacsConstraintManager`

Der YACS Constraint-Manager (YCM) dient als zentrale Schnittstelle des YACS-Frameworks. Er initiiert das Auslesen der Constraint-Lösungsstrategien und verwaltet die entsprechenden Constraint-Netze. Über eine Instanz des YCM lassen sich Constraints anhand des Namens einer verfügbaren Constraint-Lösungsstrategie inkrementell zu den Constraint-Netzen hinzugefügen. Der YCM initiiert und steuert die Anwendung der Constraint-Lösungsstrategien und führt den phasenweisen Propagations- bzw. Lösungsprozess durch.

Deklaration:

- `public interface YacsConstraintManager`

Methoden:

- `public List getStrategyNames()`

Liefert eine „unmodifizierbare“ Liste mit den Namen (`Strings`) der existierenden Constraint-Lösungsstrategien und entsprechend der verfügbaren Constraint-Netze.

- `public List getConstraintNets()`

  Liefert eine „unmodifizierbare“ Liste mit verfügbaren Constraint-Netzen (enthält Instanzen der Klasse `yacs.net.ConstraintNet`).

- `public ConstraintNet getConstraintNet(String strategyName)`
  `throws StrategyNotFoundException`

  Liefert das entsprechende Constraint-Netz zu dem übergebenen Strategienamen, `null` wenn kein Constraint-Netz mit dieser Strategie existiert.

- `public void addConstraint(String constraint, String strategyName)`
  `throws ConstraintParserException, StrategyNotFoundException`

  Mit dieser Methoden kann ein Constraint inkrementell dem Lösungsprozess zugeführt werden. Neben dem eigentlichen Constraint-Ausdruck als `String` wird der Name der Strategie benötigt, mit der das Constraint verarbeitet werden soll.

- `public void addConstraints(String constraint, String strategyName)`
  `throws ConstraintParserException, StrategyNotFoundException`

  Mit dieser Methoden können mehrere Constraints gleichzeitig bzw. ein Teilproblem inkrementell dem Lösungsprozess zugeführt werden. Neben dem eigentlichen Constraint-Ausdruck als `String` (die Constraint-Ausdrücke müssen darin jeweils durch ein Semikolon voneinander getrennt sein) wird der Name der Strategie benötigt, mit der die Constraints verarbeitet werden sollen.

- `public void setVariableDomain(String variableName, Domain domain)`
  `throws VariableNotFoundException`

  Setzt den Wertebereich für eine Variable.

- `public Domain getVariableDomain(String variableName)`
  `throws VariableNotFoundException`

  Liefert die entsprechende Domäne zur übergebenen Variable.

- `public List getVariables()`

  Liefert eine Liste mit den Namen aller Variablen sämtlicher vorhandenen Constraint-Netze (enthält Instanzen der Klasse `String`).

- `public List getVariables(String strategyName)`
  `throws StrategyNotFoundException`

  Liefert eine Liste mit den Variablen (Instanzen der Klasse `yacs.parser.Variable`), die sich in dem der übergebenen Strategie zugehörigen Constraint-Netz befinden.

- `public Solutions getSolutions(String strategyName)`
  `throws StrategyNotFoundException`

  Liefert die ggf. bereits gefundenen Lösungen für das der übergebenen Strategie zugehörige Teilproblem.

- `public boolean hasSolutions(String strategyName)`
  `throws StrategyNotFoundException`

  Liefert `true`, wenn für das der übergebenen Strategie zugehörige Teilproblem bereits Lösungen gefunden wurden, `false` wenn nicht.

- `public boolean hasFalseDomain(String strategyName)`
  `throws StrategyNotFoundException`

  Liefert `true`, wenn die Domäne einer Variablen des der übergebenen Strategie zugehörigen Teilproblems leer ist. In diesem Fall ist für dieses Teilproblem keine Konsistenz bzw. Lösung herstellbar. Ansonsten wird `false` zurückgegeben.

- `public boolean hasValuationDomain(String strategyName)`
  `throws StrategyNotFoundException`

  Liefert `true`, wenn für das der übergebenen Strategie zugehörige Teilproblem eine *valuation domain* vorliegt, d. h. in den Domänen aller Variablen des Teilproblems ist jeweils genau ein Wert enthalten. Ansonsten wird `false` zurückgegeben.

- `public boolean isInconsistent(String strategyName)`
  `throws StrategyNotFoundException`

  Gibt zurück, ob ein Constraint-Solver für das der übergebenen Strategie zugehörige Teilproblem eine Inkonsistenz gemeldet hat oder nicht (wird zurückgesetzt durch die benutzerinitiierte Modifikation der Wertebereiche der Constraint-Variablen und durch Hinzufügen einer Lösung zum Teilproblem).

- `public void evaluate()`

  Startet den Constraint-Lösungsvorgang. Ruft der Reihe nach die jeweiligen Auswertemethoden der vorhandenen Constraint-Netze auf.

- `public void tabularasa()`

  Setzt sämtliche Constraint-Netze wieder in den Ausgangszustand zurück, d. h. es werden alle Informationen über die zugehörigen Teilprobleme einer Strategie (Constraints, Variablen, Domänen, etc.) gelöscht.

E.1.2 Klasse `YacsConstraintManagerImpl`

Implementierung des YACS Constraint-Managers.

Deklaration:

- `public class YacsConstraintManagerImpl`
  `implements YacsConstraintManager`

Konstruktoren:

- `public YacsConstraintManagerImpl()`

  Wird dieser Konstruktor ohne Angabe des Pfades zu der XML-Datei mit den Constraint-Lösungsstrategien aufgerufen, wird diese im aktuellen Verzeichnis unter dem Namen `yacs_strategies.xml` gesucht.

- `public YacsConstraintManagerImpl(String strategyPath)`

  Der Parameter `strategyPath` ist der Pfad zur der XML-Datei, in der die Constraint-Lösungsstrategien spezifiziert sind.

E.2 Package `yacs.net`

Enthält die Klassen, die zum Aufbau und zur Verwaltung von Constraint-Netzen und deren zugehörigen Strategien notwendig sind.

E.2.1 Klasse `ConstraintNet`

Klasse zur Repräsentation des Constraint-Netzes eines Teilproblems innerhalb von YACS.

Deklaration:

- `public class ConstraintNet`

Konstruktoren:

- `ConstraintNet(Strategy strategy)`

  Das Constraint-Netz muss mit einer zugehörigen Strategie instantiiert werden, die für die Verarbeitung der Constraints verwendet wird.

- `ConstraintNet(Strategy strategy, Expression expr)`

  Benötigt die zugehörige Strategie und ein oder mehrere Constraints als `Expression` (ein Constraint-Teilproblem generiert aus einem `String` mit Kommata-separierter Liste von Constraint-Ausdrücken).

Methoden:

- `public void addConstraint(Expression constraint)`

  Fügt ein Constraint dem vorhandenen Constraint-`Expression` hinzu. Damit ein einheitlicher Namensraum sichergestellt ist, muss durch den Parser sichergestellt werden, dass neue Variablen mit den Namen bereits existierender Variablen auf dassselbe Objekt verweisen.

- `public String getStrategyName()`

  Liefert den Namen der zu diesem Teilproblem gehörigen Constraint-Lösungsstrategie.

- `public Expression getSubproblem()`

  Liefert das Constraint-Teilproblem bzw. den Constraint-Ausdruck.

- `public List getConstraints()`

  Liefert eine Liste der in diesem Teilproblem enthaltenen (primitiven) Constraints (enthält Instanzen der Klasse `yacs.parser.Expression`).

- `public List getVariables()`

  Liefert eine „unmodifizierbare“ Liste aller Variablen (als `Strings`) des Teilproblems.

- `public Solutions getSolutions()`

  Liefert die bereits gefundenen Lösungen für dieses Teilproblem.

- `public boolean addSolution(Solution solution)`

  Fügt diesem Teilproblem eine Lösungen hinzu. Liefert `true`, wenn dies erfolgreich verlaufen ist (Lösung war noch nicht vorhanden), `false` wenn nicht (Lösung existierte bereits)

- `public List getDomains()`

  Erzeugt und liefert eine „unmodifizierbare“ Liste aller Domänen (Instanzen der der Klasse `yacs.domain.Domain`) des Teilproblems.

- `public Domain getDomain(String variableName)`

  Liefert die entsprechende Domäne zur übergebenen Variable. Gibt `null` zurück, wenn keine Variable unter dem angegebenen Namen existiert.

- `public boolean setDomain(String variableName, Domain variableDomain)`

  Belegt die angegebene Variable mit der übergebenen Domäne.

- `public boolean hasFalseDomain()`

  Liefert `true`, wenn die Domäne einer Variable des Teilproblems leer ist. In diesem Fall ist keine Konsistenz bzw. Lösung herstellbar. Ansonsten wird `false` zurückgegeben.

- `public boolean hasValuationDomain()`

  Liefert `true`, wenn eine *valuation domain* vorliegt, d. h. in den Domänen aller Variablen des Teilproblems ist jeweils genau ein Wert enthalten. Ansonsten wird `false` zurückgegeben.

- `public boolean hasSolutions()`

  Liefert `true`, wenn für das Teilproblem bereits Lösungen gefunden wurden, `false` wenn nicht.

- `public boolean isInconsistent()`

  Gibt zurück, ob von einem Constraint-Solver eine Inkonsistenz gemeldet wurde (durch Auslösen einer `InconsistencyException`).

- `public boolean triggerPreprocessing()`

  Phase 1 der Constraint-Verarbeitung: Wendet die in der Strategie spezifizierten Preprozessing-Solver auf das Constraint-Netz an. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

- `public boolean triggerConsistency()`

  Phase 2 der Constraint-Verarbeitung: Wendet die in der Strategie spezifizierten Solver zur Konsistenzherstellung auf das Constraint-Netz an. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

- `public boolean triggerSearch()`

  Phase 3 der Constraint-Verarbeitung: Wendet die in der Strategie spezifizierten Solver zur Lösungssuche auf das Constraint-Netz an. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

- `public void clear()`

  Initialisiert das Constraint-Netz, d. h. es wird der `Expression` für das Teilproblem (inkl. der Variablen und Domänen sowie der primitiven Constraints), die bisher gefundenen Lösungen sowie die Informationen über die Konsistenz des Constraint-Netzes gelöscht.

- `public String toString()`

  Rückgabe des Constraint-Netzes als `String` (für Debugging).

E.2.2 Klasse `Solution`

Klasse zur Repräsentation einer Lösung eines Constraint-Problems. Eine Lösung besteht aus der Menge der Variablen des Problems und einer jeweils zugeordneten Wertebelegung.

Deklaration:

- `public class Solution`

Konstruktoren:

- `public Solution()`

Methoden:

- `public void addVariable(String variableName, DomainElement value)`

  Fügt die übergebene Variable (`String`) mit der angegebenen Wertebelegung (Instanz des Interface `yacs.domain.DomainElement`) dieser Lösung hinzu.

- `public List getVariables()`

  Liefert eine „unmodifizierbare" Liste mit den Namen (`Strings`) der an dieser Lösung involvierten Variablen.

- `public DomainElement getValue(String variableName)`

  Liefert die Wertebelegung der übergebenen Variable in dieser Lösung.

- `public boolean equals(Solution solution)`

  Überprüft, ob die übergebene Lösung identisch mit der aktuellen ist.

- `public String toString()`

  Generiert zu Debugging-Zwecken einen `String` mit den Variablen/Wertebelegungen dieser Lösung.

E.2.3 Klasse `Solutions`

Container-Klasse für die Repräsentation von Lösungen eines Constraint-Problems.

Deklaration:

- `public class Solutions`

Konstruktoren:

- `public Solutions()`

Methoden:

- `public boolean addSolution(Solution solution)`

  Fügt die übergebene Lösung den vorhandenen hinzu, wenn sie nicht bereits enthalten ist. Liefert `true`, wenn dies erfolgreich verlaufen ist (Lösung war noch nicht vorhanden), `false` wenn nicht (Lösung existierte bereits).

- `public List getSolutions()`

  Liefert eine „unmodifizierbare" Liste mit den enthaltenen Lösungen (Instanzen der Klassse `yacs.net.Solution`).

- `public boolean hasSolutions()`

  Liefert `true`, wenn Lösungen existieren, `false` wenn nicht.

- `public boolean containsSolution(Solution solution)`

  Überprüft, ob die übergebene Lösung bereits enthalten ist (`true`) oder nicht (`false`).

- `public String toString()`

  Generiert zu Debugging-Zwecken einen `String` mit den in diesem Container enthaltenen Lösungen.

E.2.4 Klasse `Strategy`

Klasse zur Repräsentation einer Constraint-Lösungsstrategie von YACS.

Deklaration:

- `public class Strategy`

Konstruktoren:

- `public Strategy(String strategyName)`

  Benötigt den Namen der Strategie.

Methoden:

- `public String getStrategyName()`

  Liefert den Namen der Constraint-Lösungsstragie.

- `public List getPreprocessingSolvers()`

  Liefert die Liste der in dieser Strategie spezifizierten Constraint-Solver zum Preprozessing (enthält Instanzen der Klasse `yacs.Solver.PreprocessingSolver`).

- `public List getConsistencySolvers()`

  Liefert die Liste der in dieser Strategie spezifizierten Constraint-Solver zur Konsistenzherstellung (enthält Instanzen der Klasse `yacs.Solver.ConsistencySolver`).

- `public List getSearchSolvers()`

  Liefert die Liste der in dieser Strategie spezifizierten Constraint-Solver zur Lösungssuche (enthält Instanzen der Klasse `yacs.Solver.SearchSolver`).

- `public String toString()`

  Rückgabe der Strategie als `String` (für Debugging).

E.2.5 Klasse `StrategyReader`

XML-Parser, der aus der übergebenen Datei die Constraint-Lösungsstrategien ausliest und die entsprechende Objekte (Instanzen der Klasse `yacs.net.Strategy`) instantiiert.

Deklaration:

- `public class StrategyReader`

Konstruktoren:

- `public StrategyReader(String strategyPath)`

  Der Parameter `strategyPath` ist der Pfad zur der XML-Datei, in der die Constraint-Lösungsstrategien spezifiziert sind.

Methoden:

- `public List getStrategies()`

  Liefert die eingelesenen Strategien als Liste (enthält Instanzen der Klasse `Strategy`).

E.3 Package `yacs.parser`

Enthält die vom Parser generierte, rekursive Struktur, mit der ein Constraint-Ausdruck repräsentiert wird.

E.3.1 Abstrakte Klasse `Expression`

Abstrakte Klasse für eine rekursive Struktur zur Repräsentation eines algebraischen Constraint-Problems.

Deklaration:

- `public abstract class Expression`
  `implements Cloneable, Serializable`

Methoden:

- `public abstract boolean setDomain(String variableName, Domain domain)`

  Belegt den Wertebereich der angegebenen Variable und gibt zurück, ob dies erfolgreich war (`true`) oder nicht (`false`).

- `public abstract Domain getDomain(String variableName)`

  Liefert die Domäne der übergebenen Variable.

- `public abstract List getVariables()`

  Liefert eine Liste der Variablennamen des Constraint-Problems.

- `public abstract List getUnsetVariables()`

  Liefert eine Liste mit den Namen der noch unbelegten Variablen.

- `public abstract boolean unsetVariablesLeft()`

  Gibt zurück ob unbelegte Variablen existieren (`true`) oder nicht (`false`).

- `public abstract boolean set(String variableName, DomainElement element)`

  Belegt die übergebene Variable mit einem Wert. War die Wertebelegung erfolgreich, wird `true` zurückgegeben, ansonsten `false`.

- `public abstract DomainElement get(String variableName)`

  Liefert die Wertebelegung der übergebenen Variable.

- `public abstract boolean satisfiable()`

  Prüft die Erfüllbarkeit des Constraint-Ausdrucks bzgl. der aktuellen Wertebelegung.

- `public abstract DomainElement evaluate()`

  Wertet den Constraint-Ausdruck aus. Liefert als Ergebnis die Belegung einer Variable, den Wert einer Konstante oder das Ergebnis einer Berechnung.

- `public abstract List getPrimitiveExpressions()`

  Liefert die einzelnen „Primärausdrücke" des Constraint-Problems innerhalb einer Liste (enthält Instanzen des Interface `Expression`).

- `public abstract Variable getVariableReference(String variableName)`

  Liefert die Referenz auf die Variable mit dem übergebenen Namen.

- `public abstract boolean containsVariable(String variableName)`

  Liefert `true`, wenn der `Expression` die Variable mit dem übergebenen Namen enthält. Ansonsten wird `false` zurückgegeben.

- `public abstract boolean hasFalseDomain()`

  Liefert `true`, wenn die Domäne einer Variable leer ist. In diesem Fall ist keine Konsistenz bzw. Lösung herstellbar. Ansonsten wird `false` zurückgegeben.

- `public abstract boolean hasValuationDomain()`

  Liefert `true`, wenn eine *valuation domain* vorliegt, d. h. in den Domänen aller Variablen ist jeweils genau ein Wert enthalten. Ansonsten wird `false` zurückgegeben.

- `public abstract boolean setValuationDomain()`

  Wenn eine *valuation domain* vorliegt, werden die Variablen mit dem in der jeweiligen Domäne einzig vorhandenen Wert belegt. Liefert `true`, wenn das Ergebnis positiv ist, d. h. die Werte einer *valuation domain* konnten alle gesetzt werden. Ansonsten wird `false` zurückgegeben.

- `public abstract void singletons()`

  Entfernt bei allen Variablen, die mit einem Wert belegt sind, sämtliche anderen Elemente aus deren Domäne. Variablen, die nicht mit einem Wert belegt sind, bleiben unverändert.

- `public abstract int count(variableName)`

  Liefert den „Vernetzungsgrad" einer Variablen innerhalb des Constraint-Problems, d. h. dem Grad ihres Vorkommens in den Constraints.

- `public abstract void init()`

  Initialisiert die Constraint-Variablen. Es werden alle vorhandenen Belegungen entfernt.

- `public abstract String toString()`

  Konvertiert das Objekt in einen `String`.

- `public Object clone()`

  Liefert eine Kopie dieses Objekts. Allerdings nur eine *shallow copy*, d. h. die Felder des Objektes werden nicht geklont.

- `public Expression cloneDeep()`

  Klont `this` und rekursiv alle darin referenzierten Objekte mit (die Selbheit indirekt mehrfach referenzierter Objekte bleibt erhalten).

E.3.2 Klasse `BinaryOperator`

Klasse zur Repräsentation eines binären Operators innerhalb einer rekursiven Struktur zur Repräsentation eines algebraischen Constraint-Problems (Dokumentation siehe abstrakte Klasse `yacs.parser.Expression`).

Deklaration:

- `public class BinaryOperator`
  `extends Expression`

Konstruktoren:

- `public BinaryOperator(int type, String name, Expression expr1, Expression expr2)`

  Instantiiert den Operator mit seinem Typ, seinem „Namen" und den beiden zugehörigen Constraint-Ausdrücken.

Methoden:

- `public String refine()`

  Liefert den Namen der Variable, deren Wertebereich eingeschränkt wurde. Gibt `null` zurück, wenn keine Änderung vorgenommen wurde. ACHTUNG: Verarbeitet zurzeit ausschließlich Constraints bzw. `Expressions`, die eine *solution function* darstellen.

E.3.3 Klasse `UnaryOperator`

Klasse zur Repräsentation eines unären Operators innerhalb einer rekursiven Struktur zur Repräsentation eines algebraischen Constraint-Problems (Dokumentation siehe abstrakte Klasse `yacs.parser.Expression`).

Deklaration:

- `public class UnaryOperator`
  `extends Expression`

Konstruktoren:

- `public UnaryOperator(int type, String name, Expression expr)`

  Instantiiert den Operator mit seinem Typ, seinem „Namen“ und dem zugehörigen Constraint-Ausdruck.

E.3.4 Klasse `Variable`

Klasse zur Repräsentation einer numerischen Variablen innerhalb einer rekursiven Struktur zur Repräsentation eines algebraischen Constraint-Problems (Dokumentation siehe abstrakte Klasse `yacs.parser.Expression`).

Deklaration:

- `public class Variable`
  `extends Expression`

Konstruktoren:

- `public Variable(String name)`

  Instantiiert die Variable mit ihrem Namen.

- `public Variable(String name, Domain domain)`

  Instantiiert die Variable mit ihrem Namen und ihrer Domäne.

- `public Variable(String name, Domain domain, DomainElement element)`

  Instantiiert die Variable mit ihrem Namen, ihrer Domäne und einer aktuellen Belegung.

Methoden:

- `public String getName()`

  Liefert den Namen der Variablen.

- `public Domain getDomain()`

  Liefert die zugehörige Domäne der Variablen. Diese Methode ermöglicht die direkte Abfrage (anstatt der `getDomain(String)`-Methode) unter Umgehung der rekursiven Struktur, wenn sichergestellt ist, dass das vorhandene Objekt eine Instanz der Klasse `Variable` ist.

- `public boolean setValue(DomainElement element)`

  Belegt diese Variable mit einem Element aus ihrer Domäne. Liefert `true`, wenn dies erfolgreich war, `false` wenn nicht (Wert des Elements in der Domäne nicht vorhanden). Diese Methode ermöglicht den direkten Aufruf (anstatt der `set()`-Methode) unter Umgehung der rekursiven Struktur, wenn sichergestellt ist, dass das vorhandene Objekt eine Instanz der Klasse `Variable` ist.

- `public DomainElement getValue()`

  Liefert die Wertebelegung der Variable. Diese Methode ermöglicht die direkte Abfrage (anstatt der `get()`-Methode) unter Umgehung der rekursiven Struktur, wenn sichergestellt ist, dass das vorhandene Objekt eine Instanz der Klasse `Variable` ist.

- `public boolean deleteValue(DomainElement element)`

  Entfernt das übergebene Element aus der Domäne der Variablen. Ist die Variable mit diesem Element aktuell belegt, so ist sie anschließend unbelegt (`null`). Liefert `true`, wenn das Entfernen erfolgreich war (Element war in der Domäne enthalten), `false` wenn nicht.

- `public boolean narrow(int operatorType, DomainElement element)`

  Liefert `true`, wenn der Wertebereich der Variable eingeschränkt wurde. Gibt `false` zurück, wenn keine Änderung vorgenommen wurde.

E.3.5 Klasse `Constant`

Klasse zur Repräsentation einer Konstante innerhalb einer rekursiven Struktur zur Repräsentation eines algebraischen Constraint-Problems (Dokumentation siehe abstrakte Klasse `yacs.parser.Expression`).

Deklaration:

- `public class Constant`
  `extends Expression`

Konstruktoren:

- `public Constant(DomainElement element)`

  Instantiierung der Konstante mit ihrer Belegung.

- `public Constant(Integer value)`

  Aus Gründen der Vereinfachung kann zur Instantiierung auch ein `Integer`-Objekt angegeben werden. Das benötigte `DomainElement` bzw. `NumericFDElement` wird entsprechend erzeugt.

- `public Constant(int value)`

  Aus Gründen der Vereinfachung kann zur Instantiierung auch lediglich ein `int`-Wert angegeben werden. Das benötigte `DomainElement` bzw. `NumericFDElement` wird entsprechend erzeugt.

- `public Constant(Double lo, Double hi)`

  Aus Gründen der Vereinfachung können zur Instantiierung auch zwei `Double`-Objekte als untere und obere Schranke eines Intervalls angegeben werden. Das benötigte `DomainElement` bzw. `IntervalDomainElement` wird entsprechend erzeugt.

- `public Constant(double lo, double hi)`

  Aus Gründen der Vereinfachung können zur Instantiierung auch lediglich zwei `double`-Werte als untere und obere Schranke eines Intervalls angegeben werden. Das benötigte `DomainElement` bzw. `IntervalDomainElement` wird entsprechend erzeugt.

E.4 Package `yacs.domain`

Enthält die benötigten Klassen zur Repräsentation unterschiedlicher Wertebereiche von Constraint-Variablen.

E.4.1 Klasse `Domain`

Klasse zur Repräsentation der Domäne einer Constraint-Variablen.

Deklaration:

- `public class Domain`
  `implements Cloneable, Serializable`

Konstruktoren:

- `public Domain()`

  Instantiiert die Domäne.

- `public Domain(DomainElement element)`

  Die Domäne wird mit einem Element instantiiert.

- `public Domain(List domain)`

  Die Domäne wird mit einer Liste mit Elementen instantiiert (Instanzen des Interface `yacs.domain.DomainElement`).

Methoden:

- `public boolean add(DomainElement element)`

  Der Domäne wird ein Element hinzugefügt, wenn sein Wert noch nicht vorhanden ist. Liefert `true` zurück, wenn dies erfolgreich war, `false` wenn nicht (Wert bereits vorhanden).

- `public boolean update(DomainElement element)`

  Aktualisiert den Wert für das übergebene Element in der Domäne.

- `public List getElements()`

  Liefert eine „unmodifizierbare“ Liste mit den Elementen der Domäne (Instanzen des Interface `yacs.domain.DomainElement`).

- `public DomainElement getElement(DomainElement element)`

  Liefert eine Referenz auf das in dieser Domäne vorhandene Element, welches der Belegung des übergebenen Elements entspricht. Liefert `null`, wenn kein Element mit der entsprechenden Belegung existiert.

- `public boolean deleteElement(DomainElement element)`

  Löscht eine Element aus der Domäne. Sollte nur aus der Klasse Variable aufgerufen werden, denn u.U. ist der zu löschende Wert die aktuelle Belegung der Variable. Dies wird in Variable abgefangen. Liefert `true`, wenn das Entfernen erfolgreich war (Element war in der Domäne enthalten), `false` wenn nicht.

- `public boolean containsElement(DomainElement element)`

  Abfragemöglichkeit, ob ein bestimmtes Element in der Domäne enthalten ist.

- `public boolean isEmpty()`

  Liefert `true` wenn die Domäne leer ist, `false` wenn nicht.

- `public int size()`

  Liefert die Anzahl der Elemente in der Domäne.

- `public boolean narrow(int operatorType, DomainElement element)`

  Führt eine Beschränkung der Domäne durch, so dass sie nur noch das übergebene Element enthält. Liefert `true` wenn die Domäne eingeschränkt wurde (Element war enthalten), `false` wenn nicht. (Anm.: Die Angabe `operatorType` (bezogen auf den Operator: `=`, `!=`, `<`, `>`, `<=`, `>=`) wird derzeit an dieser Stelle ignoriert.)

- `public String toString()`

  Liefert die Domäne als `String`.

- `public Object clone()`

  Liefert eine Kopie dieses Objekts. Allerdings nur eine *shallow copy*, d. h. die Felder des Objektes werden nicht geklont.

- `public Domain cloneDeep()`

  Klont `this` und rekursiv alle darin referenzierten Objekte mit (die Selbheit indirekt mehrfach referenzierter Objekte bleibt erhalten).

E.4.2 Abstrakte Klasse `DomainElement`

Abstrakte Klasse zur Repräsentation von einzelnen Werten einer Domäne einer Constraint-Variable. Implementierungen dieser Klasse können z. B. Elemente für numerische finite Domänen, symbolische finite Domänen oder reellwertige Intervall-Domänen sein.

Deklaration:

- `public abstract class DomainElement`
  `implements Comparable, Cloneable, Serializable`

Konstruktoren:

- `public DomainElement()`

Methoden:

- `public abstract Object getValue()`

  Liefert den Wert des Elements.

- `public abstract DomainElement plus(DomainElement param)`

  Methode zur Addition zweier Elemente.

- `public abstract DomainElement minus(DomainElement param)`

  Methode zur Subtraktion zweier Elemente.

- `public abstract DomainElement times(DomainElement param)`

  Methode zur Multiplikation zweier Elemente.

- `public abstract DomainElement divide(DomainElement param)`

  Methode zur Division zweier Elemente.

- `public abstract DomainElement uminus()`

  Liefert das Element mit negativem Vorzeichen.

- `public abstract boolean equals(DomainElement param)`

  Liefert `true`, wenn das Element gleich dem übergebenen Element ist, `false` wenn nicht.

- `public abstract boolean notEquals(DomainElement param)`

  Liefert `true`, wenn das Element ungleich dem übergebenen Element ist, `false` wenn nicht.

- `public abstract boolean greaterAs(DomainElement param)`

  Liefert `true`, wenn das Element größer als das übergebenen Element ist, `false` wenn nicht.

- `public abstract boolean lowerAs(DomainElement param)`

  Liefert `true`, wenn das Element kleiner als das übergebenen Element ist, `false` wenn nicht.

- `public abstract boolean greaterEqualAs(DomainElement param)`

  Liefert `true`, wenn das Element größer/gleich dem übergebenen Element ist, `false` wenn nicht.

- `public abstract boolean lowerEqualAs(DomainElement param)`

  Liefert `true`, wenn das Element kleiner/gleich dem übergebenen Element ist, `false` wenn nicht.

- `public abstract String toString()`

  Liefert die Belegung dieses Elements als `String`.

- `public int compareTo(Object param)`

  Bietet die Möglichkeit Instanzen dieser Klasse innerhalb einer TreeMap sortieren zu lassen. Hierfür ist außerdem erforderlich, dass diese Klasse das Interface `Comparable` implementiert.

- `public Object clone()`

  Liefert eine Kopie dieses Objekts. Allerdings nur eine *shallow copy*, d. h. die Felder des Objektes werden nicht geklont.

- `public DomainElement cloneDeep()`

  Klont `this` und rekursiv alle darin referenzierten Objekte mit (die Selbheit indirekt mehrfach referenzierter Objekte bleibt erhalten).

E.4.3 Klasse `IntervalDomain`

Klasse zur Repräsentation einer reellwertigen Intervalldomäne. Diese Klasse stützt sich auf die in der Bibliothek IAMath von Timothy J. Hickey implementierte Repräsentation von Intervallen.

Deklaration:

- `public class IntervalDomain`
  `extends Domain`

Konstruktoren:

- `public IntervalDomain()`

  Instantiiert eine Intervalldomäne.

- `public IntervalDomain(RealInterval value)`

  Die Domäne wird mit einem Wert (hier: `RealInterval`) instantiiert.

- `public IntervalDomain(Double lo, Double hi)`

  Erlaubt die vereinfachte Instantiierung einer Intervalldomäne durch Angabe der unteren (`lo`) und oberen Schranke (`hi`) des Intervalls als `Double`-Objekt.

- `public IntervalDomain(double lo, double hi)`

  Erlaubt die vereinfachte Instantiierung einer Intervalldomäne durch Angabe der unteren (`lo`) und oberen Schranke (`hi`) des Intervalls als `double`-Wert.

Methoden:

- `public boolean add(DomainElement value)`

  Der Domäne wird ein Element hinzugefügt. Überschreibt die übergeordnete `add()`-Methode in der Art, dass sichergestellt wird, dass das übergebene Element anschließend das einzige innerhalb der Domäne ist. Dadurch wird ein konvexes Intervall repräsentiert. Die Methode liefert `true` zurück, wenn die Belegung erfolgreich war, `false` wenn nicht.

- `public boolean add(RealInterval value)`

  Vereinfacht das Hinzufügen von Elementen zu Domäne, indem lediglich ein Real-Intervall übergeben werden muss. Liefert `true` zurück, wenn dies erfolgreich war, `false` wenn nicht (Intervall bereits vorhanden). Diese Methode stellt sicher, dass das übergebene Intervall anschließend das einzige Element innerhalb der Domäne ist, wodurch ein konvexes Intervall repräsentiert wird.

- `public boolean add(Double lo, Double hi)`

Vereinfacht das Hinzufügen von Elementen zu Domäne, indem lediglich die untere (`lo`) und obere Schranke (`hi`) des Intervalls als `Double`-Objekt angegeben werden muss. Liefert `true` zurück, wenn dies erfolgreich war, `false` wenn nicht (Intervall bereits vorhanden). Diese Methode stellt sicher, dass das übergebene Intervall anschließend das einzige Element innerhalb der Domäne ist, wodurch ein konvexes Intervall repräsentiert wird.

- `public boolean add(double lo, double hi)`

  Vereinfacht das Hinzufügen von Elementen zu Domäne, indem lediglich die untere (`lo`) und obere Schranke (`hi`) des Intervalls als `double`-Wert angegeben werden muss. Liefert `true` zurück, wenn dies erfolgreich war, `false` wenn nicht (Intervall bereits vorhanden). Diese Methode stellt sicher, dass das übergebene Intervall anschließend das einzige Element innerhalb der Domäne ist, wodurch ein konvexes Intervall repräsentiert wird.

- `public double getLowerBound()`

  Liefert die untere Schranke dieser Intervalldomäne.

- `public double getUpperBound()`

  Liefert die obere Schranke dieser Intervalldomäne.

- `public void setLowerBound(double lowerBound)`

  Setzt die untere Schranke dieser Intervalldomäne.

- `public void setUpperBound(double upperBound)`

  Setzt die obere Schranke dieser Intervalldomäne.

- `public boolean narrow(int operatorType, DomainElement element)`

  Führt eine Beschränkung der Domäne durch, so dass sich das enthaltene Intervall anschließend innerhalb den Grenzen des übergebenen Intervalls befindet. Lässt sich diese Beschränkung nicht durchführen, wird das in dieser Domäne enthaltenen Intervall gelöscht, wodurch eine *false domain* und dadurch eine Inkonsistenz entsteht. Liefert `true` wenn die Domäne eingeschränkt wurde, `false` wenn nicht. (Anm.: Derzeit wird auf diese Art ausschließlich das Einschränken der Intervallgrenzen für den Gleichheitsoperator implementiert.)

E.4.4 Klasse `IntervalDomainElement`

Klasse zur Repräsentation von einzelnen „Elementen" für reellwertige Intervall-Domänen. Mit einem Element ist in diesem Fall ein kontinuierliches (Teil-)Intervall gemeint. Diese Klasse stützt sich auf die in der Bibliothek IAMath von Timothy J. Hickey implementierte Intervallarithmetik.

Deklaration:

- `public class IntervalDomainElement`
  `extends DomainElement`

Konstruktoren:

- `public IntervalDomainElement(RealInterval value)`

  Benötigt ein `RealInterval`-Objekt zur Instantiierung.

- `public IntervalDomainElement(Double lo, Double hi)`

  Aus Gründen der Vereinfachung können zur Instantiierung auch zwei `Double`-Objekte als untere und obere Schranke des Intervalls angegeben werden.

- `public IntervalDomainElement(double lo, double hi)`

  Aus Gründen der Vereinfachung können zur Instantiierung auch lediglich zwei `double`-Werte als untere und obere Schranke des Intervalls angegeben werden.

Methoden:

- `public double getLowerBound()`

  Liefert die untere Schranke dieses Intervalls.

- `public double getUpperBound()`

  Liefert die obere Schranke dieses Intervalls.

- `public void setLowerBound(double lowerBound)`

  Setzt die untere Schranke dieses Intervalls.

- `public void setUpperBound(double upperBound)`

  Setzt die obere Schranke dieses Intervalls.

- `public boolean contains(DomainElement param)`

  Liefert `true`, wenn alle Werte des übergebenen Intervalls in diesem Intervall vollständig enthalten sind, `false` wenn nicht.

E.4.5 Klasse `NumericFDDomain`

Klasse zur Repräsentation einer finiten `Integer`-Domäne.

Deklaration:

- `public class NumericFDDomain`
  `extends Domain`

Konstruktoren:

- `public NumericFDDomain()`

  Instantiiert eine Domäne mit numerischen FD-Elementen.

- `public NumericFDDomain(Integer value)`

  Die Domäne wird mit einem Wert (hier: `Integer`) instantiiert.

- `public NumericFDDomain(int value)`

  Die Domäne wird mit einem Wert (hier: `int`) instantiiert.

- `public NumericFDDomain(int startValue, int endValue)`

  Erlaubt die vereinfachte Instantiierung einer FD-Domäne mit einem Wertebereich „von" (`startValue`) – „bis" (`endValue`).

Methoden:

- `public boolean add(Integer value)`

  Vereinfacht das Hinzufügen von Elementen zu Domäne, indem lediglich ein `Integer`-Objekt übergeben werden muss. Liefert `true` zurück, wenn dies erfolgreich war, `false` wenn nicht (Wert bereits vorhanden).

- `public boolean add(int value)`

  Vereinfacht das Hinzufügen von Elementen zu Domäne, indem lediglich ein `int`-Wert übergeben werden muss. Liefert `true` zurück, wenn dies erfolgreich war, `false` wenn nicht (Wert bereits vorhanden).

- `public void add(int startValue, int endValue)`

  Der Domäne wird ein Bereich „von" (`startValue`) – „bis" (`endValue`) hinzugefügt.

- `public boolean contains(int value)`

  Vereinfachte Abfragemöglichkeit, ob ein Element mit einem bestimmten `int`-Wert in der Domäne enthalten ist.

E.4.6 Klasse `NumericFDElement`

Klasse zur Repräsentation von einzelnen Elementen für numerische finite Domänen. Der Wert eines Elements wird durch eine Instanz der Klasse Integer zum Ausdruck gebracht.

Deklaration:

- `public class NumericFDElement`
  `extends DomainElement`

Konstruktoren:

- `public NumericFDElement(Integer value)`

  Benötigt ein `Integer`-Objekt zur Instantiierung.

- `public NumericFDElement(int value)`

  Aus Gründen der Vereinfachung kann zur Instantiierung auch lediglich ein `int`-Wert angegeben werden.

E.4.7 Klasse `SymbolicFDDomain`

Klasse zur Repräsentation einer symbolischen finiten Domäne.

Deklaration:

- `public class SymbolicFDDomain`
  `extends Domain`

Konstruktoren:

- `public SymbolicFDDomain()`

  Instantiiert eine Domäne mit symbolischen FD-Elementen.

- `public SymbolicFDDomain(String value)`

  Die Domäne wird mit einem Wert (hier: `String`) instantiiert.

Methoden:

- `public boolean add(String value)`

  Vereinfacht das Hinzufügen von Elementen zu Domäne, indem lediglich ein `String` übergeben werden muss. Liefert `true` zurück, wenn dies erfolgreich war, `false` wenn nicht (Wert bereits vorhanden).

- `public boolean contains(String value)`

  Vereinfachte Abfragemöglichkeit, ob ein Element mit einem bestimmten Wert (hier: `String`) in der Domäne enthalten ist.

E.4.8 Klasse `SymbolicFDElement`

Klasse zur Repräsentation von einzelnen Elementen für symbolische finite Domänen. Der Wert eines Elements wird durch eine Instanz der Klasse `String` zum Ausdruck gebracht. Die Methoden dieser Klasse implementieren Operationen auf `Strings`, wie das Anhängen durch `plus()` oder Entfernen durch `minus()` eines Substrings an bzw. aus einem bestehenden `String`. Die Methoden `times()` und `divide()` sowie `uminus()` sind „Dummies“, die nur ein SymbolicFDElement mit der Belegung “” (leerer `String`) bzw. das bestehende Objekt zurückgeben. Die Vergleichsoperatoren entsprechen lexikographischen `String`-Vergleichen (`equals()` und `compareTo()`).

Deklaration:

- `public class SymbolicFDElement`
  `extends DomainElement`

Konstruktoren:

- `public SymbolicFDElement(String value)`

  Benötigt ein `String`-Objekt zur Instantiierung.

E.5 Package yacs.solver

Enthält die Constraint-Solver bzw. die Constraint-Lösungsalgorithmen von YACS.

E.5.1 Interface Solver

Interface für Constraint-Solver innerhalb von YACS.

Deklaration:

- `public interface Solver`

Methoden:

- `public boolean evaluate(ConstraintNet constraintNet)`
  `throws InconsistencyException`

  Methode zur Initiierung des Lösungsprozesses des jeweiligen Constraint-Solvers. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

E.5.2 Abstrakte Klasse ConsistencySolver

Abstrakte Klasse für Constraint-Solver mit Verfahren zur Herstellung eines speziellen Konsistenzgrades.

Deklaration:

- `public abstract class ConsistencySolver`
  `implements Solver`

Methoden:

- `public abstract boolean propagate(ConstraintNet constraintNet) throws InconsistencyException`

  Benötigt als Eingabe ein Constraint-Netz. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

E.5.3 Abstrakte Klasse `PreprocessingSolver`

Abstrakte Klasse für Constraint-Solver mit Preprozessing-Verfahren.

Deklaration:

- `public abstract class PreprocessingSolver implements Solver`

Methoden:

- `public abstract boolean process(ConstraintNet constraintNet) throws InconsistencyException`

  Benötigt als Eingabe ein Constraint-Netz. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

E.5.4 Abstrakte Klasse `SearchSolver`

Abstrakte Klasse für Constraint-Solver mit Verfahren zur Lösungssuche.

Deklaration:

- `public abstract class SearchSolver implements Solver`

Methoden:

- `public abstract boolean search(ConstraintNet constraintNet) throws InconsistencyException`

  Benötigt als Eingabe ein Constraint-Netz. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

E.6 Package `yacs.solver.fdsolver.consistency`

Enthält Constraint-Solver zur Herstellung bestimmter Konsistenzgrade für finite Domänen.

E.6.1 Klasse `AC3Solver`

Einfacher Solver zum Herstellen von Kantenkonsistenz über finite Domänen mittels `AC-3` nach Mackworth (1977a, S. 106).

Deklaration:

- `public class AC3Solver`
  `extends ConsistencySolver`

Methoden:

- `public boolean arcSolve(Expression expr)`
  `throws OnlyBinaryConstraintsAllowedException, InconsistencyException`

  Einfacher Constraint-Solver, der Kantenkonsistenz mittels `AC-3` herstellt. `AC-3` kann ausschließlich binäre Constraints verarbeiten, d. h. es dürfen maximal zwei Variablen in jedem primitiven Ausdruck vorhanden sein! Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

E.6.2 Klasse `ACSolver`

Einfacher Algorithmus zum Herstellen von Kantenkonsistenz über finite Domänen nach Marriott und Stuckey (1999, S. 94).

Deklaration:

- `public class ACSolver`
  `extends ConsistencySolver`

Methoden:

- `public boolean arcSolve(Expression expr)`
  `throws InconsistencyException`

  Einfacher Algorithmus zum Herstellen von Kantenkonsistenz. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

E.6.3 Klasse `BinaryArc`

Hilfsklasse zur Beschreibung einer „Kante“ für binäre Algorithmen zur Herstellung von Kantenkonsistenz. Eine Kante zwischen zwei Knoten entspricht einem binären Constraint, welches zwei Constraint-Variablen beschränkt.

Deklaration:

- `public class BinaryArc`
  `extends HyperArc`

Konstruktoren:

- `public BinaryArc(String startNode, String endNode, Expression expr)`

  Benötigt den Start- und Endknoten der Kante sowie den Constraint-Ausdruck, der diese Kante beinhaltet.

Methoden:

- `public String startNode()`

  Liefert den Startknoten der Kante.

- `public Domain startDomain()`

  Liefert die Domäne des Startknotens.

- `public String endNode()`

  Liefert den Endknoten der Kanten.

- `public Domain endDomain()`

  Liefert die Domäne des Endknotens.

E.6.4 Klasse `HyperArc`

Hilfsklasse zur Beschreibung einer beliebigstelligen „Kante" für Algorithmen zur Herstellung von (Hyper-)Kantenkonsistenz. Eine Kante zwischen n Knoten entspricht einem n-stelligen Constraint, welches n Constraint-Variablen beschränkt.

Deklaration:

- `public class HyperArc`

Konstruktoren:

- `public HyperArc(List nodes, Expression expr)`

  Benötigt eine Liste mit den Knoten der Kante (enthält Instanzen der Klasse `String`) sowie den Constraint-Ausdruck, der diese Kante beinhaltet.

Methoden:

- `public List getNodes()`

  Liefert eine „unmodifizierbare" Liste mit den Knoten der Kante (enthält Instanzen der Klasse `String`).

- `public Domain getDomain(String node)`

  Liefert die jeweilige Domäne zum übergebenen Knoten.

- `public Expression expression()`

  Liefert den Constraint-Ausdruck, zu dem diese Kante gehört.

- `public String toString()`

  Liefert die Kante als `String` (für Debugging).

E.6.5 Klasse `NCSolver`

Einfacher Algorithmus zum Herstellen von Knotenkonsistenz über finite Domänen nach Marriott und Stuckey (1999, S. 94).

Deklaration:

- `public class NCSolver`
  `extends ConsistencySolver`

Methoden:

- `public boolean nodeSolve(Expression expr)`
  `throws InconsistencyException`

  Eine Solver der Knotenkonsistenz herstellt, d. h. es werden alle unären Constraints auf konsistente Wertebereiche überprüft und selbige ggf. um unzulässige Werte reduziert. Liefert `true`, wenn Wertebereichseinschränkungen vorgenommen wurden, `false` wenn nicht.

E.7 Package `yacs.solver.fdsolver.search`

Enthält Constraint-Solver zur Lösungsssuche für finite Domänen.

E.7.1 Klasse `BacktrackingSolver`

Einfacher Backtracking-Solver zur Lösungssuche in einem Constraint-Problem mit finiten Domänen nach Dechter und Frost (1998, S. 13) bzw. Dechter und Frost (2002, S. 152). Der Solver wurde dahingehend modifiziert, dass anstatt lediglich einer Lösung alle möglichen Lösungen eines Problems gefunden werden.

Deklaration:

- `public class BacktrackingSolver`
  `extends SearchSolver`

E.7.2 Klasse `DomDegRatioVariableOrdering`

Generiert auf Basis des „Fail-First"-Prinzips (FF) und der „Maximum-Degree-Ordering"-Heuristik (MDO) eine Reihenfolge für die Belegung von Variablen während einer Lösungssuche (vgl. Bessière und Régin 1996). Das Prinzip dieser Heuristik ist, dass Variablen, deren Verhältnis von Größe des Wertebereichs (`dom`) zum Vernetzungsgrad (`deg`) innerhalb des Constraint-Problems am geringsten ist, zuerst mit einem Wert belegt werden (`dom/deg`).

Deklaration:

- `public class DomDegRatioVariableOrdering`

Methoden:

- `public List getOrdering(ConstraintNet constraintNet)`

  Generiert eine `dom/deg`-Ordnung für die Variablen des übergebenen Constraint-Netzes. Liefert eine entsprechend geordnete Liste mit Variablennamen (`Strings`).

E.7.3 Klasse `MAC3Solver`

Einfacher MAC-3-Solver zur Lösungssuche mit *look-ahead* in einem Constraint-Problem mit finiten Domänen nach Dechter und Frost (2002, S. 175 u. 178). Der Solver wurde dahingehend modifiziert, dass anstatt lediglich einer Lösung alle möglichen Lösungen eines Problems gefunden werden.

Deklaration:

- `public class MAC3Solver`
  `extends SearchSolver`

E.7.4 Klasse `MACSolver`

Einfacher MAC-Solver zur Lösungssuche mit *look-ahead* in einem Constraint-Problem mit finiten Domänen nach Dechter und Frost (2002, S. 175 u. 178). Der Solver wurde dahingehend modifiziert, dass anstatt lediglich einer Lösung alle möglichen Lösungen eines Problems gefunden werden.

Deklaration:

- `public class MACSolver`
  `extends SearchSolver`

E.7.5 Klasse `SingleSolutionBTSolver`

Ein rekursiver Backtracking-Solver für finite Domänen in Anlehnung an Marriott und Stuckey (1999, S. 90). In dieser Implementierung allerdings wird nicht nur `true/false` zurückgegeben, sondern ggf. eine Lösung generiert. Dies ist die erste Lösung, die durch den Suchalgorithmus aufgefunden wird. Der Solver eignet sich in erster Linie für die Überprüfung, ob ein Problem inkonsistent ist oder nicht.

Deklaration:

- `public class SingleSolutionBTSolver`
  `extends SearchSolver`

E.8 Package `yacs.solver.intervalsolver.consistency`

Enthält Constraint-Solver zur Konsistenzherstellung für Constraint-Variablen mit reellwertigen Intervallen als Wertebereich.

E.8.1 Klasse `HullConsistencySolver`

Einfacher Algorithmus zum Herstellen von Hull-Konsistenz für Variablen mit Wertebereichen aus reellwertigen Intervallen. Der Algorithmus orientiert sich am Waltz-Filteralgorithmus und den Beschreibungen von Davis (1987), Hyvönen (1992) und Lhomme (1993). Eine Zerlegung der Constraints findet nicht statt, außerdem lassen sich ausschließlich Constraints verarbeiten, die bereits vollständig als *solution functions* (vgl. Hyvönen 1992) bzw. als „Projektionen" vorliegen (vgl. Lhomme 1993).

Deklaration:

- `public class HullConsistencySolver`
  `extends ConsistencySolver`

Methoden:

- `public boolean hullSolve(Expression expr)`
  `throws InconsistencyException`

  Einfacher Solver zum Herstellen von Hull- bzw. 2B-Konsistenz, basierend auf dem Waltz-Filteralgorithmus (in diesem Fall für Domänen mit reellwertigen Intervallen).

E.9 Package `yacs.exceptions`

In diesem Package sind eine Reihe von Exceptions enthalten, die je nach Bedarf z. B. bei der Implementierung eines Constraint-Solvers eingesetzt werden können, bzw. von den bestehenden Komponenten von YACS bereits genutzt werden.

- `ConstraintParserException`

  Diese Exception wird von YACS ausgelöst, wenn beim Parsen eines Constraints ein Fehler auftritt.

- `InconsistencyException`

  Diese Exception wird von YACS ausgelöst, wenn während der Propagation oder Lösungssuche eine Inkonsistenz auftritt.

- `OnlyBinaryConstraintsAllowedException`

  Diese Exception wird ausgelöst, wenn einer Constraint-Lösungskomponente, die ausschließlich dazu in der Lage ist, binäre Constraints zu behandeln, höherwertige Constraints zur Verarbeitung vorgelegt werden.

- `OnlyConsistencySolversAllowedException`

  Diese Exception wird ausgelöst, wenn an einer Stelle, an der ausschließlich Constraint-Solver für die Konsistenzherstellung genutzt werden sollen, fälschlicherweise andere Solver übergeben wurden.

- `OnlyPreprocessingSolversAllowedException`

  Diese Exception wird ausgelöst, wenn an einer Stelle, an der ausschließlich Constraint-Solver für die Preprozessing-Phase genutzt werden sollen, fälschlicherweise andere Solver übergeben wurden.

- `OnlySearchSolversAllowedException`

  Diese Exception wird ausgelöst, wenn an einer Stelle, an der ausschließlich Constraint-Solver für die Lösungssuche genutzt werden sollen, fälschlicherweise andere Solver übergeben wurden.

- `StrategyNotFoundException`

  Diese Exception wird ausgelöst, wenn auf eine Strategy zugegriffen werden soll, die nicht existiert.

- `StrategyParserException`

  Diese Exception wird bei einem Fehler innerhalb des XML-Parsers für die Constraint-Lösungsstrategien von YACS ausgelöst.

- `VariableNotFoundException`

  Diese Exception wird ausgelöst, wenn auf eine Variable zugegriffen werden soll, die nicht existiert.

Anhang F

Programm `YacsTester`

Das Programm `YacsTester` dient dazu, verschiedene Funktionalitäten von YACS zu testen und den Gebrauch des Frameworks zu demonstrieren. Auf das an dieser Stelle abgedruckte Programm-Listing wird in Kapitel 8 in Abschnitt 8.3 auf Seite 225 ff. zur Validierung des YACS-Frameworks anhand synthetischer Problemstellungen Bezug genommen.

Zum Programmablauf: Durch das Programm `YacsTester` wird im Konstruktor eine Initialisierung unterschiedlicher Constraint-Probleme vorgenommen und anschließend zwei Auswertevorgänge durchgeführt. Dazwischen werden Modifikationen an den bestehenden Constraint-Problemen vorgenommen. Die Ergebnisse der Auswertevorgänge werden jeweils ausgegeben.

```java
import yacs.*;
import yacs.domain.*;
import yacs.exceptions.*;
import yacs.net.*;

import java.util.*;

import org.apache.log4j.*;

/**
 * <p>Klasse zum Testen verschiedener Funktionalitaeten von YACS.</p>
 *
 * <p>Copyright (C) 2005  Wolfgang Runte
 * <br><br>
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 * <br><br>
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 * <br><br>
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
```

```
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301
 * USA</p>
 *
 * @author Wolfgang Runte (woru@tzi.org)
 * @version YACS 0.1.1
 */
public class YacsTester {

// ********** private attributs **********

  /** Logger aktivieren: */
  private static Logger logger = Logger.getLogger("yacs");

  /** Dateipfad zu den Constraint-Loesungsstrategien. */
  private final String strategyPath = "yacs_strategies.xml";

  /** Instanz des YACS Constraint-Managers. */
  private final YacsConstraintManager ycm;

// ********** public attributs **********

// ********** private methods **********

  /**
   * Ein paar Testausgaben: Der Strategiename, das zugehoerigen Teilproblem, die
   * primitiven Constraint-Ausdruecke sowie die involvierten Variablen und deren
   * Domaenen.
   * @param strategyName String
   */
  private void printTestValues(String strategyName) {
    try {
      // Constraint-Netz unter Angabe des Strategienamens vom YCM holen:
      ConstraintNet constraintNet = this.ycm.getConstraintNet(strategyName);
      // Liste der vorhandenen Variablen des Teilproblems holen:
      List variablesList = constraintNet.getVariables();
      // Testausgaben:
      System.out.println("Strategie: '" + constraintNet.getStrategyName() + "'");
      System.out.println("Expression: " + constraintNet.getSubproblem());
      List exprList = constraintNet.getConstraints();
      System.out.println("Primitive Constraints: (" + exprList.size() + ")");
      for (int i = 0; i < exprList.size(); i++) {
        System.out.println(exprList.get(i));
      }
      System.out.println("Variablen: " + constraintNet.getVariables());
      System.out.println("Domaenen der Constraint-Variablen: (" +
                         variablesList.size() + ")");
      try {
        for (int i = 0; i < variablesList.size(); i++) {
          System.out.println(variablesList.get(i) + ": " +
                             this.ycm.
                             getVariableDomain((String)variablesList.get(i)));
        }
      } catch (VariableNotFoundException e) {
        e.printStackTrace();
```

```
      }
      System.out.println("Constraint-Netz -> " +
                         this.ycm.getConstraintNet(strategyName));
      System.out.println("Inkonsistenz: " + this.ycm.isInconsistent(strategyName));
    } catch (StrategyNotFoundException e) {
      e.printStackTrace();
    }
  }

  /**
   * Constraints unter Angabe des Namens der zu verwendenen Strategie zum YACS
   * Constraint-Manager (YCM) hinzufuegen.
   * @param strategyName String
   * @param constraintArray String[]
   */
  private void initConstraints(String strategyName, String[] constraintArray) {
    for (int i = 0; i < constraintArray.length; i++) {
      try {
        this.ycm.addConstraint(constraintArray[i], strategyName);
      } catch (ConstraintParserException e) {
        e.printStackTrace();
      } catch (StrategyNotFoundException e) {
        e.printStackTrace();
      }
    }
  }

// ********** public methods **********

  /**
   * NCSolverTest
   * @param init boolean
   */
  public void NCSolverTest(boolean init) {
    List variablesList;
    // Name der Strategie:
    String strategyName = "low_consistency";
    try {
      if (init) {
        System.out.println();
        System.out.println("Einfaches Problem zum Testen von Knotenkonsistenz:");
        System.out.println("==================================================");
        // Constraints spezifizieren:
        String constraintArray[] = new String[4];
        constraintArray[0] = "x < y;";
        constraintArray[1] = "y < z;";
        constraintArray[2] = "z <= 2;";
        constraintArray[3] = "3 + y < 7;";
        // Constraints zum YCM hinzufuegen:
        this.initConstraints(strategyName, constraintArray);
        // Liste der Variablen der Strategie bzw. des Constraint-Netzes holen:
        variablesList = this.ycm.getVariables(strategyName);
        // Wertebereiche der Constraint-Variablen setzen:
        try {
```

```
          for (int i = 0; i < variablesList.size(); i++) {
            this.ycm.setVariableDomain((String)variablesList.get(i),
                                       new NumericFDDomain(0, 9));
          }
        } catch (VariableNotFoundException e) {
          e.printStackTrace();
        }
        // Testausgaben:
        this.printTestValues(strategyName);
      } else {
        System.out.println();
        System.out.println("Ergebnis fuer Knotenkonsistenz:");
        System.out.println("===============================");
        // Testausgaben:
        this.printTestValues(strategyName);
        // Ergebnis ausgeben:
        if (!this.ycm.hasFalseDomain(strategyName)) {
          System.out.println("\nErgebnis ist knotenkonsistent!");
        } else {
          System.out.println("\nKnotenkonsistenz nicht herstellbar!");
        }
      }
    } catch (StrategyNotFoundException e) {
      e.printStackTrace();
    }
  }

  /**
   * AC3SolverTest
   * @param init boolean
   */
  public void AC3SolverTest(boolean init) {
    List variablesList;
    // Name der Strategie:
    String strategyName = "medium_consistency";
    try {
      if (init) {
        System.out.println();
        System.out.println("Kartenfaerbeproblem fuer eine Australienkarte:");
        System.out.println("==============================================");
        // Constraints spezifizieren:
        String constraintArray[] = new String[9];
        constraintArray[0] = "WA != NT;";
        constraintArray[1] = "WA != SA;";
        constraintArray[2] = "NT != SA;";
        constraintArray[3] = "NT != Q;";
        constraintArray[4] = "SA != Q;";
        constraintArray[5] = "SA != NSW;";
        constraintArray[6] = "SA != V;";
        constraintArray[7] = "Q != NSW;";
        constraintArray[8] = "T = T;";
        // Constraints zum YCM hinzufuegen:
        this.initConstraints(strategyName, constraintArray);
        // Liste der Variablen der Strategie bzw. des Constraint-Netzes holen:
```

```java
        variablesList = this.ycm.getVariables(strategyName);
        // Wertebereiche der Constraint-Variablen setzen:
        SymbolicFDDomain domain;
        try {
          for (int i = 0; i < variablesList.size(); i++) {
            domain = new SymbolicFDDomain();
            domain.add(new SymbolicFDElement("rot"));
            domain.add(new SymbolicFDElement("gruen"));
            domain.add(new SymbolicFDElement("blau"));
            this.ycm.setVariableDomain((String)variablesList.get(i), domain);
          }
          // Die Einschraenkungen vornehmen, um auf Inkonsistenzen pruefen zu
          // koennen:
          this.ycm.setVariableDomain("WA", new SymbolicFDDomain("rot"));
        } catch (VariableNotFoundException e) {
          e.printStackTrace();
        }
        // Testausgaben:
        this.printTestValues(strategyName);
      } else {
        System.out.println();
        System.out.println("Ergebnis fuer Kartenfaerbeproblem:");
        System.out.println("==================================");
        // Testausgaben:
        this.printTestValues(strategyName);
        // Ergebnis ausgeben:
        if (!this.ycm.hasFalseDomain(strategyName)) {
          System.out.println("\nErgebnis ist kantenkonsistent!");
        } else {
          System.out.println("\nKantenkonsistenz nicht herstellbar!");
        }
      }
    } catch (StrategyNotFoundException e) {
      e.printStackTrace();
    }
  }

  /**
   * SingleSolutionBTSolverTest
   * @param init boolean
   */
  public void SingleSolutionBTSolverTest(boolean init) {
    List variablesList;
    // Name der Strategie:
    String strategyName = "simple_search";
    try {
      if (init) {
        System.out.println();
        System.out.println("Simple Search:");
        System.out.println("==============");
        // Constraints spezifizieren:
        String constraintArray[] = new String[2];
        constraintArray[0] = "X < Y;";
        constraintArray[1] = "Y < Z;";
```

```
          // Constraints zum YCM hinzufuegen:
          this.initConstraints(strategyName, constraintArray);
          // Liste der Variablen der Strategie bzw. des Constraint-Netzes holen:
          variablesList = this.ycm.getVariables(strategyName);
          // Wertebereiche der Constraint-Variablen setzen:
          try {
            for (int i = 0; i < variablesList.size(); i++) {
              this.ycm.setVariableDomain((String)variablesList.get(i),
                                          new NumericFDDomain(1, 2));
            }
          } catch (VariableNotFoundException e) {
            e.printStackTrace();
          }
          // Testausgaben:
          this.printTestValues(strategyName);
        } else {
          System.out.println();
          System.out.println("Ergebnis fuer Simple Search:");
          System.out.println("============================");
          // Testausgaben:
          this.printTestValues(strategyName);
          // Ergebnis ausgeben:
          try {
            if (this.ycm.hasSolutions(strategyName)) {
              System.out.println("\nLoesung(en):" +
                                 this.ycm.getSolutions(strategyName));
            } else {
              System.out.println(
                "\nEs konnte keine Loesung fuer das Problem ermittelt werden.");
            }
          } catch (StrategyNotFoundException e) {
            e.printStackTrace();
          }
        }
      } catch (StrategyNotFoundException e) {
        e.printStackTrace();
      }
    }

    /**
     * BacktrackingSolverTest
     * @param init boolean
     */
    public void BacktrackingSolverTest(boolean init) {
      List variablesList;
      // Name der Strategie:
      String strategyName = "search";
      try {
        if (init) {
          System.out.println();
          System.out.println("Smuggler's Knapsack (1):");
          System.out.println("========================");
          // Constraints spezifizieren:
          String constraintArray[] = new String[2];
```

```
        constraintArray[0] = "4*W + 3*P + 2*C <= 9;";
        constraintArray[1] = "15*W + 10*P + 7*C >=30;";
        // Constraints zum YCM hinzufuegen:
        this.initConstraints(strategyName, constraintArray);
        // Liste der Variablen der Strategie bzw. des Constraint-Netzes holen:
        variablesList = this.ycm.getVariables(strategyName);
        // Wertebereiche der Constraint-Variablen setzen:
        try {
          for (int i = 0; i < variablesList.size(); i++) {
            this.ycm.setVariableDomain((String)variablesList.get(i),
                                       new NumericFDDomain(0, 9));
          }
        } catch (VariableNotFoundException e) {
          e.printStackTrace();
        }
        // Testausgaben:
        this.printTestValues(strategyName);
      } else {
        System.out.println();
        System.out.println("Ergebnis fuer Smuggler's Knapsack (1):");
        System.out.println("=======================================");
        // Testausgaben:
        this.printTestValues(strategyName);
        // Ergebnis ausgeben:
        try {
          if (this.ycm.hasSolutions(strategyName)) {
            System.out.println("\nLoesung(en):" +
                               this.ycm.getSolutions(strategyName));
          } else {
            System.out.println(
              "\nEs konnte keine Loesung fuer das Problem ermittelt werden.");
          }
        } catch (StrategyNotFoundException e) {
          e.printStackTrace();
        }
      }
    } catch (StrategyNotFoundException e) {
      e.printStackTrace();
    }
  }

  /**
   * MAC3SolverTest
   * @param init boolean
   */
  public void MAC3SolverTest(boolean init) {
    List variablesList;
    // Name der Strategie:
    String strategyName = "search_with_look-ahead";
    try {
      if (init) {
        System.out.println();
        System.out.println("Smuggler's Knapsack (2):");
        System.out.println("========================");
```

```
        // Constraints spezifizieren:
        String constraintArray[] = new String[2];
        constraintArray[0] = "4*W + 3*P + 2*C <= 9;";
        constraintArray[1] = "15*W + 10*P + 7*C >=30;";
        // Constraints zum YCM hinzufuegen:
        this.initConstraints(strategyName, constraintArray);
        // Liste der Variablen der Strategie bzw. des Constraint-Netzes holen:
        variablesList = this.ycm.getVariables(strategyName);
        // Wertebereiche der Constraint-Variablen setzen:
        try {
          for (int i = 0; i < variablesList.size(); i++) {
            this.ycm.setVariableDomain((String)variablesList.get(i),
                                       new NumericFDDomain(0, 9));
          }
        } catch (VariableNotFoundException e) {
          e.printStackTrace();
        }
        // Testausgaben:
        this.printTestValues(strategyName);
      } else {
        System.out.println();
        System.out.println("Ergebnis fuer Smuggler's Knapsack (2):");
        System.out.println("======================================");
        // Testausgaben:
        this.printTestValues(strategyName);
        // Ergebnis ausgeben:
        try {
          if (this.ycm.hasSolutions(strategyName)) {
            System.out.println("\nLoesung(en):" +
                               this.ycm.getSolutions(strategyName));
          } else {
            System.out.println(
              "\nEs konnte keine Loesung fuer das Problem ermittelt werden.");
          }
        } catch (StrategyNotFoundException e) {
          e.printStackTrace();
        }
      }
    } catch (StrategyNotFoundException e) {
      e.printStackTrace();
    }
  }

  /**
   * HullConsistencySolverTest
   * @param init boolean
   */
  public void HullConsistencySolverTest(boolean init) {
    List variablesList;
    // Name der Strategie:
    String strategyName = "interval_consistency";
    try {
      if (init) {
        System.out.println();
```

```
            System.out.println("Einfaches Problem zum Testen von Hull-Konsistenz:");
            System.out.println("================================================");
            // Constraints spezifizieren:
            String constraintArray[] = new String[2];
            constraintArray[0] = "v1 + v2 = v3; v2 = v3 - v1; v1 = v3 - v2;";
            constraintArray[1] = "v2 = v1; v1 = v2;";
            // Constraints zum YCM hinzufuegen:
            this.initConstraints(strategyName, constraintArray);
            // Wertebereiche der Constraint-Variablen setzen:
            try {
              this.ycm.setVariableDomain("v1", new IntervalDomain(2, 12));
              this.ycm.setVariableDomain("v2", new IntervalDomain(4, 9));
              this.ycm.setVariableDomain("v3", new IntervalDomain(2, 8));
            } catch (VariableNotFoundException e) {
              e.printStackTrace();
            }
            // Testausgaben:
            this.printTestValues(strategyName);
          } else {
            System.out.println();
            System.out.println("Ergebnis fuer Hull-Konsistenz:");
            System.out.println("==============================");
            // Testausgaben:
            this.printTestValues(strategyName);
            // Ergebnis ausgeben:
            if (!this.ycm.hasFalseDomain(strategyName)) {
              System.out.println("\nErgebnis ist konsistent!");
            } else {
              System.out.println("\nKonsistenz nicht herstellbar!");
            }
          }
        } catch (StrategyNotFoundException e) {
          e.printStackTrace();
        }
      }

      /**
       * Konstruktor
       */
      public YacsTester() {
        // Den YACS Constraint-Manager mit dem Pfad zu den
        // Constraint-Loesungsstrategien instantiieren:
        this.ycm = new YacsConstraintManagerImpl(this.strategyPath);
        // Loglevel setzen (moegliche Level: ALL, DEBUG, INFO, WARN, ERROR, FATAL):
        this.logger.setLevel(Level.OFF);

        // Testprobleme initialisieren:
        this.NCSolverTest(true);
        this.AC3SolverTest(true);
        this.SingleSolutionBTSolverTest(true);
        this.BacktrackingSolverTest(true);
        this.MAC3SolverTest(true);
        this.HullConsistencySolverTest(true);

```

```
    // Constraint-Loesungsprozess initiieren:
    System.out.println("\n");
    System.out.println("*************************************");
    System.out.println("* Starten der Constraint-Auswertung! *");
    System.out.println("*************************************");
    System.out.println();
    this.ycm.evaluate();
    // Ergebnisse ausgeben:
    this.NCSolverTest(false);
    this.AC3SolverTest(false);
    this.SingleSolutionBTSolverTest(false);
    this.BacktrackingSolverTest(false);
    this.MAC3SolverTest(false);
    this.HullConsistencySolverTest(false);

    // Einige Constraints hinzufuegen:
    System.out.println("\n");
    System.out.println("Fuege zusaetzliche Constraints hinzu:");
    System.out.println("=====================================");
    try {
      this.ycm.addConstraint("z > 2;", "low_consistency");
      this.ycm.addConstraint("NSW != V;", "medium_consistency");
      this.ycm.addConstraint("W > 1;", "search");
      this.ycm.addConstraint("W > 1;", "search_with_look-ahead");
      this.ycm.addConstraint("v1 = [5,5];", "interval_consistency");
    } catch (ConstraintParserException e) {
      e.printStackTrace();
    } catch (StrategyNotFoundException e) {
      e.printStackTrace();
    }
    // Einige zusaetzliche Wertebereichseinschraenkungen vornehmen:
    System.out.println("\n");
    System.out.println("Nehme Aenderungen an den Wertebereichen vor:");
    System.out.println("============================================");
    try {
      System.out.println("Variable: 'V' -> Wert: 'blau'");
      this.ycm.setVariableDomain("V", new SymbolicFDDomain("blau"));
      System.out.println("Variable: 'W' -> Wert: '2'");
      this.ycm.setVariableDomain("W", new NumericFDDomain(2));
      System.out.println("Variable: 'Z' -> Wert: '1' bis '3'");
      this.ycm.setVariableDomain("Z", new NumericFDDomain(1, 3));
    } catch (VariableNotFoundException e) {
      e.printStackTrace();
    }
    // Constraint-Loesungsprozess initiieren:
    System.out.println("\n");
    System.out.println("***************************************************");
    System.out.println("* Nochmaliges Starten der Constraint-Auswertung! *");
    System.out.println("***************************************************");
    System.out.println();
    this.ycm.evaluate();
    // Ergebnisse ausgeben:
    this.NCSolverTest(false);
    this.AC3SolverTest(false);
```

```
      this.SingleSolutionBTSolverTest(false);
      this.BacktrackingSolverTest(false);
      this.MAC3SolverTest(false);
      this.HullConsistencySolverTest(false);
    }

    public static void main(String[] args) {
      YacsTester yacsTester = new YacsTester();
    }

}
```

Anhang G

Glossar

Algebra (arabisch) Die Lehre von den mathematischen Gleichungen und deren Strukturen.

Apache-Lizenz Die *Apache-Lizenz* ist die Freie-Software-Lizenz der *Apache Software Foundation*. Die aktuelle Version 2.0 wurde im Januar 2004 veröffentlicht.[1] Sie ist gegenüber der vorherigen Version 1.1 stark erweitert. Prinzipiell beinhaltet sie, dass Software unter dieser Lizenz frei in jedem Umfeld verwendet, modifiziert und verteilt werden darf. Wenn Software verteilt wird, muss eindeutig darauf hingewiesen werden, welche Software unter der *Apache-Lizenz* verwendet wurde und dass diese von *Apache Software Foundation* stammt. Eine Kopie der Lizenz muss dem Paket beiliegen. Änderungen am Quellcode der *Apache-Software* müssen nicht zu *Apache* zurückgeschickt werden. Die eigene Software, die *Apache-Software* verwendet, muss nicht unter der *Apache-Lizenz* stehen.

API (Abkürzung für engl. *Application Programming Interface*) Eine *API* ist eine dokumentierte Software-Schnittstelle, mit deren Hilfe ein Programm die Funktionen eines anderen Programms nutzen kann (gleiches gilt für die *API* eines Betriebssystems).

Arithmetik (griechisch) „Zahlenlehre“; arithmetisch: das Rechnen mit Zahlen betreffend.

Backus-Naur-Form (BNF) Die Backus-Naur-Form ist eine kompakte und formale Metasyntax, die zur Darstellung von *kontextfreie Grammatiken* eingesetzt wird. Dies betrifft die Syntax gängiger höherer Programmiersprachen. Sie wird auch für die Notation von Befehlssätzen und Kommunikationsprotokollen verwendet. Durch die *Backus-Naur-Form* ist es möglich, die Syntax einer Programmiersprache formal exakt, d. h. ohne die Ungenauigkeiten natürlicher Sprachen, darzustellen.

Binärbaum Als Binärbaum bezeichnet man in der Graphentheorie eine spezielle Form eines Graphen. Es handelt sich in diesem Fall um einen Baum, bei dem jeder Knoten höchstens zwei Kindknoten besitzt.

[1] `http://www.apache.org/licenses/LICENSE-2.0`

Blocks World Eine Modellwelt, die in der Wahrnehmungsforschung und bei Problemlöse-, und Planungsalgorithmen verwendet wird. Es gibt nur bestimmte idealisierte Objekte, bspw. alle →*euklidischen Körper* bzw. idealisierte Operatoren (z. B. `lege_auf(A, B)`).

Branch & Bound (engl. *branch* = verzweigen, *bound* = beschränken) Methode zur Bestimmung der Lösung eines Problems, die aus dem *Backtracking* abgeleitet ist. Diese Methode liefert stets eine optimale Lösung, allerdings kann die Laufzeit exponentiell mit der Länge der Eingabe wachsen. In der Praxis hängt ihre Qualität von der Güte der Zielfunktion ab. Ein Minimierungsproblem wird dabei nach gewissen Regeln schrittweise in Teilprobleme zerlegt (*branch*). Die einzelnen Teilprobleme bilden einen Baum. Jedem Knoten im Baum wird ein Wert zugeordnet, der eine untere Schranke (*bound*) für den Wert der Zielfunktion des Optimierungsproblem sein muss. Es werden zunächst diejenigen Zweige bearbeitet, die den kleinstmöglichen Zielfunktionswert erwarten lassen. Nach Bearbeitung eines Zweiges werden alle diejenigen Zweige desselben Teilbaumes vom Gesamtbaum entfernt, deren Schranke oberhalb des bereits bekannten besten Wertes der Zielfunktion liegt (vgl. Claus und Schwill 2001).

BSD-Lizenz (Abkürzung für engl. *Berkeley-Source-Distribution*-Lizenz) Die *BSD-Lizenz* ist wie die *GNU General Public License* (→*GPL*) eine Lizenz für freie Software.[2] Ursprünglich wurde die Lizenz nur für Software verwendet, die an der Universität von Kalifornien in Berkeley entwickelt wurde, fand aber bald recht weite Verbreitung. Sie ähnelt im Wesentlichen der *GPL*, ist teilweise jedoch liberaler formuliert. So ist es wie in der *GPL* erlaubt, Software beliebig zu kopieren und zu verändern, jedoch darf das Programm auch in kommerzieller Software verwendet werden. Berühmtestes Beispiel ist die Verwendung des Netzwerkmoduls von *BSD* in Microsoft Windows.

Clustering Unter *Clustering* bzw. *Clusteranalyse* versteht man verschiedene automatische Verfahren der Datenanalyse zur Ermittlung von Gruppen (*Cluster*) zusammengehöriger Objekte aus einer Grundmenge von numerisch beschriebenen Objekten. Die Objekte können beispielsweise Datensätze von Messwerten oder Bildpunkte sein, in denen geordnete Ansammlungen oder Hierarchien gefunden werden sollen. Verfahren der *Clusteranalyse* werden vorwiegend z. B. zur automatischen Klassifikation, zur Erkennung von Mustern in der Bildverarbeitung und zum *Data-Mining* eingesetzt.

Constraint (algebraisches) Gleichung oder Ungleichung, welche die möglichen Belegungen von Variablen einschränkt. Sind bspw. zwei Variablen v_1 und v_2 auf der Menge der Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 definiert, so schränken die Gleichungen $v_1+v_2 = 10$ und $v_1 - v_2 = 2$ die Werte für v_1 und v_2 auf die Belegung $v_1 = 6$ und $v_2 = 4$ ein. Anstelle numerischer *Constraints* können auch symbolische *Constraints* verwendet werden (→*Constraint-System*).

[2] `http://www.opensource.org/licenses/bsd-license.php`

Constraint-Propagation Fortpflanzung von Beschränkungen in einem →*Constraint-System*. Charakterisiert die Arbeitsweise eines solchen Systems. Belegungen einzelner Variablen üben Beschränkungen aufeinander aus (→*Constraint*), die zu Wertebereichseinschränkungen in den Domänen der jeweiligen Variablen führen. Diese Einschränkungen breiten sich durch wiederkehrende *Propagation* über eine Vielzahl Variablen aus.

Constraint-System Ein System, welches Problemlösungen durch schrittweise Einschränkung (→*Constraint-Propagation*) der möglichen Belegungen von Variablen erzeugt (→*Constraint*).

CORBA (Abkürzung für engl. *Common Object Request Broker Architecture*) ist die Spezifikation einer Architektur, die die Definition der Schnittstellen in verteilten Systemen sowie die Kommunikation zwischen diesen Schnittstellen ermöglicht. Entwickelt wurde *CORBA* unter der Kontrolle der *Object Management Group* (OMG)[3], einer non-profit Organisation mit über 800 Mitgliedern. Die OMG erarbeitet Spezifikationen zur Verteilung und Kommunikation zwischen Objekten. Zentrale Instanz einer *CORBA*-basierten Umgebung ist der *Object Request Broker* (ORB). Implementierungen von *CORBA* sind systemunabhängige, nicht an eine bestimmte Programmiersprache gebundene und „netzwerktransparente“ (Netz-)Objektmanagementsysteme, d. h. mit *CORBA* implementierte Objekte können über die Grenzen von Betriebssystemen und Netzwerke hinaus verwendet werden. Ein Client-Programm muss dabei nicht am selben Rechner ausgeführt werden wie das *CORBA*-Objekt selbst.

Deduktion (lateinisch *deducere*, „herabführen“) Die *Deduktion* oder deduktive Methode ist in der Philosophie und der Logik eine Schlussfolgerungsweise *vom Allgemeinen auf das Besondere*, vom Vielen auf das Eine. Genauer werden mit Hilfe der *Deduktion* spezielle Einzelerkenntnisse aus allgemeinen Theorien gewonnen. Die moderne mathematische Logik und alle formalen Systeme enthalten nur *deduktive* Prinzipien. Auch die gesamte Mathematik liegt vollständig in *deduktivem* Aufbau vor und wird vorwiegend so gelehrt. Jedoch werden in der Entwicklung der Mathematik viele ihrer Erkenntnisse *induktiv* gewonnen. Gegensatz: →*Induktion*.

Determiniertheit Ein Algorithmus kann als Abbildung von der Menge der möglichen Eingabewerte in die Menge der möglichen Ausgabewerte aufgefasst werden. Ist diese Abbildung eine Funktion, d. h. bildet sie jeden möglichen Eingabewert auf höchstens einen Ausgabewert ab, so nennt man den Algorithmus *determiniert* (vgl. Claus und Schwill 2001).

Determinismus Ein Algorithmus oder Programm wird durch Maschinen schrittweise abgearbeitet. Der Algorithmus heißt *deterministisch*, wenn es zu jeder Programmsituation höchstens eine nachfolgende Situation geben kann, wenn also zu jedem Zeitpunkt der Folgeschritt eindeutig bestimmt ist. Ein deterministischer Algorithmus ist stets *determiniert*. Die Umkehrung gilt nicht. Stochastische Algorithmen sind ein Beispiel

[3] `http://www.omg.org`

für *nichtdeterministische* (und meist auch *nichtdeterminierte*) Algorithmen. Viele Probleme der Praxis lassen sich durch *nichtdeterministische* Algorithmen knapper und klarer lösen, als durch *deterministische*. Da Computer jedoch *deterministisch* arbeiten, müssen letztendlich *deterministische* Algorithmen entwickelt werden (vgl. Claus und Schwill 2001).

disjunkt Zwei Mengen heißen *disjunkt* oder *elementfremd*, wenn sie kein gemeinsames Element besitzen.

diskret unstetig, in endlichen Intervallen; Eine *diskrete* Menge ist eine Menge mit *abzählbaren* Werten. *Diskret* ist z. B. die Menge der ganzen Zahlen. Gegensatz: *kontinuierlich*.

Enterprise Java Beans (EJB) sind eine Komponentenschnittstelle für die Erstellung plattformneutraler Client/Server-Anwendungen im Internet und Intranet unter Java. *EJB* bietet als →*API* wiederverwendbare, serverseitige Komponenten. Das Framework liefert eine feste Infrastruktur zur Bereitstellung von systemnahen Diensten (Namensvergabe, Transaktionen, Messaging, Sicherheit), die eine einfache Implementierung von Geschäftsprozessen ermöglicht.

Entscheidungsbaum Methode zur grafischen Darstellung und Bearbeitung von Entscheidungsproblemen: die Methode ähnelt dem *Backtracking*. Ein *Entscheidungsbaum* zu einem Entscheidungsproblem ist ein geordneter, gerichteter Baum. Jeder Knoten des Baumes mit Ausnahme der Blätter enthält seinerseits ein Entscheidungsproblem, das Teil des Gesamtproblems ist. Diese Knoten nennt man daher *Entscheidungsknoten*. Die Kanten zwischen einem *Entscheidungsknoten* und seinen Söhnen repräsentieren mögliche Entscheidungsalternativen für das zugehörige Enscheidungsproblem. Die Blätter enthalten die Ergebnisse der jeweiligen Entscheidungsfolgen. Dabei kann es sich um Situationen handeln, die man nach der entsprechenden Entscheidungsfolge erreicht, oder um Aktionen, die man befolgen muss. Die Auswertung eines *Entscheidungsbaums* beginnt bei der Wurzel. Solange man noch kein Blatt erreicht hat, löst man das Entscheidungsproblem des gerade betrachteten Knotens durch Wahl einer der möglichen Alternativen und verzweigt zum entsprechenden Sohn. Erreicht man ein Blatt, so befolgt man gegebenenfalls die dort vorgeschriebene Aktion (vgl. Claus und Schwill 2001).

Entscheidungstabelle Sprachkonstrukt zur übersichtlichen und wartungsfreundlichen Darstellung verschachtelter bedingter Anweisungen. *Entscheidungstabellen* bestehen aus einem Entscheidungsteil und einem Aktionsteil, denen eine Werte- bzw. eine Funktionsmatrix zugeordnet ist. Es darf immer höchstens eine Spalte der Wertematrix geben, die mit der Spalte der ausgewerteten Bedingungen übereinstimmt. Die *Entscheidungstabelle* muss also die durchzuführenden Aktionen eindeutig bestimmen; sie ist deterministisches Sprachelement. Weiterhin sollte eine *Entscheidungstabelle vollständig* sein, d. h. es sollte zu jeder möglichen Spalte von Wahrheitswerten auch eine übereinstimmende Spalte in der Wertematrix geben (vgl. Claus und Schwill 2001).

euklidischer Körper Geometrischer Körper, der nur aus ebenen Flächen besteht, bspw. Quader, Würfel oder Pyramiden, nicht allerdings Zylinder oder Kugeln.

extensionale Repräsentation Explizite Aufzählung aller Elemente einer Menge. Eignet sich insbesondere für Abhängigkeiten mit relativ wenigen Elementen. Ein Constraint kann *extensional* beschrieben werden, indem alle zulässigen Belegungen der Constraint-Variablen des Constraints in einer Menge *explizit* aufgezählt werden. Gegensatz: →*intensionale Repräsentation*.

Framework (objektorientiertes) Eine Menge kooperierender Klassen, welche die Elemente eines wiederverwendbaren Entwurfs für eine bestimmte Art von Software darstellen. Ein *Framework* bietet eine Architekturhilfe beim Aufteilen des Entwurfs in abstrakte Klassen und beim Definieren ihrer Zuständigkeiten und Interaktionen. Ein Entwickler passt das *Framework* für eine bestimmte Anwendung an, indem er Unterklassen der *Framework*-Klassen bildet und ihre Objekte zusammensetzt (vgl. Gamma et al. 1996, S. 445).

Fuzzy-Logik Unscharfe Logik; eine dem menschlichen Denken angepasste, mathematische Technik, die nicht nach der für den Computer üblichen binären Logik mit Ja/Nein-Zuständen arbeitet. Die *Fuzzy-Logik* kann daher mit nicht klar definierten, relativen Zuständen arbeiten, die über den Grad ihrer Wahrscheinlichkeit erfasst werden. Sie ist eine durch empirische Daten unterstützte Arbeitsweise, die auch mit unpräzisen Eingabedaten arbeiten kann. Daher wird sie überall da verwendet, wo unklare Zustände oder Veränderungen erfasst und bearbeitet werden müssen.

GPL (Abkürzung für engl. *GNU General Public License*) bezeichnet eine Lizenz für „freie“ Software (bzw. genauer für den Programmquellcode), die von *Richard Stallman*, dem Begründer des GNU-Projekts entworfen wurde.[4] Freie Software bedeutet in diesem Zusammenhang, dass sie von jedem gebraucht, geändert und angepasst werden kann. Besonders wichtig ist, dass jegliche veränderte Software wieder unter die *GPL* gestellt werden muss, d. h. frei weitergegeben werden muss. Andere wichtige Lizenzen sind die *Lesser General Public License* (→*LGPL*) und die →*BSD-Lizenz*.

Heuristik Lehre von den möglichst erfolgreich arbeitenden Lösungsverfahren. In der Informatik: Bezeichnung für ein Lösungsverfahren, das nur zum Teil auf wissenschaftlich gesicherten Erkenntnissen, sondern vorwiegend auf Hypothesen, Analogien oder Erfahrungen aufbaut. Die Güte solcher Verfahren ist deshalb meistens nicht beweisbar, sondern wird durch wiederholte Experimente an typischen Problemstellungen nachgewiesen. Viele für die Praxis wichtigen Aufgabenstellungen gehören z. B. zu den NP-vollständigen Problemen, d. h. das Auffinden optimaler Lösungen wächst nach heutiger Kenntnis exponentiell mit der Länge der Eingabe und ist somit nicht realisierbar. Bei solchen Problemen verwendet man daher Strategien (*Heuristiken* genannt), die auf Vermutungen, plausiblen Annahmen und Erfahrungen beruhen und die i. A. relativ schnell recht gute Lösungen erzeugen. Faustregeln, stückweises

[4] `http://www.gnu.org/licenses/gpl.html`

Vortasten an eine Lösung, die Nachbildung der menschlichen Vorgehensweise zur Lösung komplexer Probleme, der Natur abgeschaute Verfahren und viele Verfahren zum Durchmustern von Entscheidungsbäumen sind Heuristiken (vgl. Claus und Schwill 2001).

HTML (Abkürzung für engl. *Hypertext Markup Language*) Standardisierte Seitenbeschreibungssprache für WWW-Seiten im Internet bzw. Intranet, welche von Charles F. Goldfarb entwickelt wurde und in der ISO-Norm 8879 definiert ist (auch →*XML*). Sie definiert sowohl die Gestaltung, den Inhalt und die Grafik der Seite als auch die Hyperlinks zu eigenen oder fremden Seiten.

Induktion Die *Induktion* bzw. *induktives* Schließen bezeichnet in der Logik und den Naturwissenschaften das Schließen vom Besonderen auf das Allgemeine zum Zweck des Erkenntnisgewinns. Im Gegensatz zur →*Deduktion* ist diese Vorgehensweise nur unter bestimmten Voraussetzungen gerechtfertigt, da Verallgemeinerungen mit Unsicherheiten behaftet sind.

Inferenz (lateinisch *inferre*, „einbringen, beitragen, folgern") Wissen, das durch logische Schlussfolgerungen gewonnen wurde

intensionale Repräsentation Implizite Angabe der charakteristischen Eigenschaft der Elemente einer (Lösungs-)Menge. Die Relation eines Constraints kann *intensional* über eine Gleichung bzw. Ungleichung definiert werden. Gegensatz: →*extensionale Repräsentation*.

JAR Das *Java Archive (JAR)* ist ein plattformunabhängiges Dateiformat, welches zum Archivieren und Komprimieren von Dateien, ähnlich den Zip-Archiven unter Windows und den Tar-Archiven unter Unix, dient. Üblicherweise werden innerhalb einer *JAR*-Datei mehrere class-Dateien zusammengefasst. Ein *JAR* enthälten jeweils eine Manifest-Datei mit Metadaten über das Archiv. *JARs* können betriebssystemunabhängig in Java genutzt werden.

Java-Applet Ein *Java-Applet* ist ein in Java geschriebenes Programm, welches in einer →*HTML*-Seite eingebettet ist. Das Programm wird heruntergeladen und vom Web-Browser ausgeführt, der über ein entsprechendes *Java-Plugin* verfügen muss. *Java-Applet* werden für dynamische Funktionen in Web-Seiten verwendet. Aus Sicherheitsgründen ist der Funktionsumfang von *Java-Applet* stark beschränkt (keine Datenzugriffe auf lokale Speichermedien etc.). Sie werden innerhalb einer *Sandbox* genannten „virtuellen Maschine" ausgeführt.

kanonische Lösung Eine Lösung wird *kanonisch* genannt, wenn sie aus Intervallen besteht, deren Grenzen entweder jeweils dieselben oder direkt aufeinander folgende Zahlen sind, d. h. wenn sie möglichst punktgenaue Lösungen darstellen.

Kartesisches Produkt Das *kartesische Produkt* zweier Mengen A und B ist die Menge aller geordneten Paare (a, b) mit $a \in A$ und $b \in B$. In Formeln:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

Analog ist $A \times B \times C$ die Menge aller Tripel aus Elementen der drei Mengen A, B und C. Das *kartesische Produkt* wird z. B. dazu benützt, um die Ebene (Zeichenebene) als kartesisches Produkt zweier Geraden (Zahlengeraden) zu konstruieren. Mathematisch gesehen ist die Ebene die Menge aller reellen Zahlenpaare, und es gilt $R^2 = R \times R$. Analog gilt für den dreidimensionalen Raum $R^3 = R \times R \times R$, und auch höherdimensionale Räume (allgemein spricht man von R^n) können so definiert werden (vgl. Embacher und Oberhuemer 2003).

Konvergenz Annäherung, Hinneigung; Bezeichnet bei einer Folge oder Reihe das Vorhandensein eines Grenzwerts.

LGPL (Abkürzung für engl. *GNU Lesser General Public License*) ist eine etwas entschärfte Variante (engl. *lesser*, „weniger“) der →*GPL*, deren Hauptunterschied darin liegt, dass die Verwendung von Programmen, die unter dieser Lizenz stehen, nicht dazu führen muss, dass die ganze Software unter dieser Lizenz (und damit frei) herausgegeben werden muss.[5] Besonders gut eignet sich diese Lizenz daher für Bibliotheken, was der alte Name, *Library General Public License*, ausdrückt. Dieser wurde jedoch geändert, da diese Lizenz nicht nur auf Bibliotheken beschränkt gelten sollte.

Maschinelles Lernen bezeichnet den Einsatz automatischer Verfahren um neue Informationen aus Daten zu gewinnen. *Maschinelles Lernen* ist ein Oberbegriff für die „künstliche“ Generierung von Wissen aus Erfahrung: Ein künstliches System lernt aus Beispielen und kann nach Beendigung der Lernphase verallgemeinern. Das heißt es lernt nicht einfach die Beispiele auswendig, sondern es „erkennt“ Gesetzmäßigkeiten in den Lerndaten. So kann das System auch unbekannte Daten beurteilen. Die Gesetzmäßigkeiten werden dabei meist nicht explizit bekannt.

Microsoft Word Nondeterministisches Textadventure-Game für Fortgeschrittene.

Mediator Die Aufgabe eines *Mediators* ist die Integration von existierenden, heterogenen Datenquellen. In der Regel verwendet ein *Mediator* dafür unterschiedliche →*Wrapper*.

Monotonie einer Funktion bezeichnet die Eigenschaft einer reellen Funktion, mit wachsendem Argument größere oder kleinere Funktionswerte anzunehmen. Eine Funktion ist *monoton* wachsend bzw. steigend, wenn $x < y \Rightarrow f(x) \leq f(y)$, sie ist streng bzw. strikt *monoton* wachsend/steigend, wenn $x < y \Rightarrow f(x) < f(y)$. Der Graph einer solchen Funktion „steigt“ mit wachsendem x „nach oben“ an. Analog gilt dies für (streng) *monoton* fallend.

Multi-Agenten-System System aus mehreren gleichartigen oder unterschiedlich spezialisierten *Agenten*, die gemeinsam eine Aufgabe bearbeiten und dazu miteinander kommunizieren.

[5] `http://www.gnu.org/licenses/lgpl.html`

Newton-Verfahren zur Lösung von Gleichungen ist eine Methode zum näherungsweisen Auffinden der Nullstellen einer differenzierbaren Funktion f. Beginnend mit einem Schätzwert x_0 für eine Nullstelle werden Schritt für Schritt gemäß der Formel

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

weitere Näherungswerte x_1, x_2, ... berechnet. Unter gewissen Voraussetzungen konvergiert diese Folge gegen die gesuchte Nullstelle. Das Verfahren hat eine einfache geometrische Interpretation: Die Tangente an den Graphen von f im Punkt $(x_n, f(x_n))$ wird bis zur x-Achse verfolgt, um den nächsten Schätzwert $x_n + 1$ zu bestimmen (vgl. Embacher und Oberhuemer 2003).

NP Problemklasse für die Menge der mit einem nichtdeterministischen Algorithmus mit polynomialen Aufwand lösbaren Probleme ($\rightarrow P$) (vgl. Claus und Schwill 2001).

NP-hart Ein Problem X ist *NP-hart*, wenn jedes Problem aus *NP* polynomial auf X reduzierbar ist (vgl. Claus und Schwill 2001).

NP-vollständig Ein Problem ist *NP-vollständig*, wenn es *NP-hart* ist, und wenn es zu *NP* gehört (vgl. Claus und Schwill 2001).

Numerik Die *numerische Mathematik*, kurz *Numerik* genannt, beschäftigt sich mit der Konstruktion und Analyse von Algorithmen für kontinuierliche mathematische Probleme.

Numerisches Lösen einer Gleichung Nicht jede Gleichung lässt sich durch eine einfache Rechnung lösen. Manchmal muss man sich mit einer näherungsweisen („numerischen", „approximativen") Lösung zufrieden geben. Da jede Gleichung (in einer Variablen) in die Form $f(x) = 0$ gebracht werden kann (wobei f eine Funktion ist), ist das Problem, sie zu lösen, gleichbedeutend damit, die Nullstellen der Funktion f zu finden. Dafür stehen etliche näherungsweise Methoden zur Verfügung. Geometrisch betrachtet, besteht das Problem darin, die Schnittpunkte des Graphen von f mit der x-Achse zu ermitteln. Der praktischste und einfachste Weg, dies zu tun, besteht darin, die Zoom-Funktion eines Funktionsplotters zu benützen, um die x-Koordinaten der Schnittpunkte mit vernünftiger Genauigkeit abzulesen (graphisches Lösen einer Gleichung). Unter *numerischen* Techniken im eigentlichen Sinn versteht man (computerunterstützte) Algorithmen, die sich (in der Regel rekursiv) zu immer höherer Genauigkeit der Lösung hinaufarbeiten. Beispiele sind die Bisektionsmethode und das Newton-Verfahren (vgl. Embacher und Oberhuemer 2003).

ODBC (Abkürzung für engl. *Open DataBase Connectivity*) Standardisierte Methode, die den $\rightarrow$*SQL*-Zugriff auf Datenbanken erlaubt, ohne dabei zu berücksichtigen, aus welchem Programm oder von welchem Betriebssystem aus der Zugriff erfolgt. *ODBC* beruht auf einer Spezifikation, die von der SQL-ACCESS-Group (SAG) unter Federführung der Firma Microsoft ins Leben gerufen wurde.

P Bezeichnung für die Menge aller Probleme, die ein deterministischer Algorithmus mit polynomialem Zeitaufwand löst ($\rightarrow NP$) (vgl. Claus und Schwill 2001).

Plugin (von engl. *to plug in*, „einstöpseln“, „anschließen“) oder Ergänzungs- oder Zusatzmodul ist eine gängige Bezeichnung für ein Softwareprogramm, das in ein anderes Softwareprodukt „eingeklinkt“ wird. Softwarehersteller definieren Schnittstellen zu ihren Produkten, mit deren Hilfe Dritte Erweiterungen – *Plugins* genannt – für diese Softwareprodukte programmieren können. Das Plugin erweitert die Funktionalität dieses Softwareprodukts.

Polymorphie benennt die Eigenschaft objektorientierter Programmiersprachen, dass in einer Klasse die geerbten Methoden redefiniert, d. h. *überschrieben* oder *überladen* werden können. *Überschreiben* bedeutet, dass in der abgeleiteten Klasse eine Methode mit dem gleichen Namen und der gleichen Signatur wie in der Oberklasse definiert wird. Die Signatur ist durch den Rückgabewert und die Parameterliste der Methode definiert. Man spricht von einer *Überladung*, wenn eine Methode mit gleichem Namen aber unterschiedlicher Parameterliste eingeführt wird.

Prädikat In der Interpretation einer formalen Sprache eines *Prädikatenkalküls* in der *Prädikatenlogik* (Erweiterung der *Aussagenlogik)* wird einem jeden *Prädikatensymbol* $P(,,\ldots)$ eine Relation P in der Menge aller Individuen (Gegenstände, Entitäten) der in dieser Interpretation betrachteten Welt zugeordnet. Dabei werden einstellige *Prädikatensymbole* auch einstelligen Relationen, zweistellige *Prädikatensymbole* zweistelligen Relationen usw. zugeordnet. Außerdem kann einem jedem n-stelligen *Prädikatensymbol* $P(,,\ldots)$ ein *Prädikat*, d. h. eine Funktion $P(x_1, x_2, \ldots, x_n)$ aus der Menge des kartesischen Produkts in die Menge der Wahrheitswerte $\{true, false\}$ zugeordnet werden, so dass $P(x_1, x_2, \ldots, x_n) = Wahr$ genau dann, wenn für die $P(,,\ldots)$ zugeordnete Relation P gilt: $(x_1, x_2, \ldots, x_n) \in P$.

Prädikat-Gleichung Definierte *Prädikate* wie z. B. $P(n)$ können gültig ($true$) oder ungültig ($false$) sein. Die logische Verknüpfung von *Prädikaten* in Gleichungen bzw. Funktionen mittels entsprechender Operatoren ($<, >, <=, >=, =$) führt wiederum zu einem gültigen oder ungültigen Ergebnis.

Propagation $\rightarrow$*Constraint-Propagation.*

Raumschiff Enterprise Fortbewegungsmittel für interdisziplinär-intergalaktische Forschungsgemeinschaften ab Ende des 21. Jahrhunderts.

SAT (von engl. *satisfiability problem*) Erfüllbarkeitsproblem der *Aussagenlogik*; Auffinden einer erfüllenden Belegung für eine *aussagenlogische Formel.*

Semantic Web Weiterentwicklung des WWW in Richtung sprachlicher Bedeutung von Inhalten. Im *Semantic Web* werden Informationen mit einer wohldefinierten Bedeutung versehen. Ziel ist es, die Kooperation zwischen Mensch und Maschine zu verbessern.

Smalltalk ist eine objektorientierte Programmiersprache, eine große Klassenbibliothek und eine interaktive Programmierumgebung in einem. *Smalltalk* war die erste konsequent objektorientierte Sprache.

Socket-Server Ein *Socket-Server* ist ein Programm, mit dem sich über beliebige Ports eines Rechners Verbindungen zu Programmen auf entfernten (oder desselben) Rechners aufbauen lassen.

SOAP (Abkürzung für engl. *Simple Object Access Protocol*) →*XML* ist ein Standard für Metadaten. Bei Nutzung von →*Web Services* erfolgt der Zugriff auf *XML*-Objekte über *SOAP*. Der Standard definiert, wie Transaktionen via Internet und *XML* getätigt sowie dynamische *Web Services* über verteilte Netzwerke genutzt werden können.

SQL (Abkürzung für engl. *Structured Query Language*) Weit verbreitete Sprache zur Definition und Manipulation relationaler Datenbanken. *SQL* wurde in den 70er-Jahren unter dem Namen *SEQUEL* von der Firma IBM entwickelt (vgl. Claus und Schwill 2001).

stetig heißt eine reelle Funktion, wenn kleine Änderungen des Arguments kleine Änderungen des Funktionswerts zur Folge haben. Der Graph einer *stetigen* Funktion ist eine zusammenhängende Kurve, (die sozusagen mit dem Bleistift nachgezogen werden kann, ohne ihn abzusetzen). Ist eine Funktion in mehreren Intervallen definiert (wie z. B. $\frac{1}{x}$, was ja für $x = 0$ nicht existiert), so macht der Begriff der *Stetigkeit* nur in Bezug auf jeden einzelnen dieser Bereiche Sinn. (Für $\frac{1}{x}$ sind das die beiden Intervalle $x < 0$ und $x > 0$). Er kann auch auf Funktionen in mehreren Variablen und Funktionen, die auf der Menge der komplexen Zahlen definiert sind, verallgemeinert werden. Gegensatz: →*unstetig* (vgl. Embacher und Oberhuemer 2003).

Stochastik (griechisch) Verfahren zur Ermittlung von Wahrscheinlichkeiten; eine an der Wahrscheinlichkeit orientierte Betrachtungsweise in den Naturwissenschaften und in der Statistik; besagt, dass bei Massenerscheinungen Aussagen nicht ganz exakt gemacht werden können, sondern nur unter Berücksichtigung gewisser zufälliger Abweichungen vom empirisch ermittelten Mittelwert.

Truth Maintenance System (TMS) „Begründungsverwaltungs-System"; System zum *nichtmonotonen Schließen*, bei dem die Wahrheitswerte von Aussagen stets an aktuelle Belege für diese Aussagen angepasst werden.

unstetig heißt eine reelle Funktion, die nicht →*stetig* ist. Die einfachsten *unstetigen* Funktionen haben Sprungstellen. An diesen ist die Funktion zwar definiert, der Graph ist aber „auseinandergerissen" (also keine zusammenhängende Kurve). Kleine Änderungen des Arguments können große Änderungen des Funktionswerts zur Folge haben (vgl. Embacher und Oberhuemer 2003).

Web Services Bei diesen Web-Diensten handelt es sich um verteilte Software-Lösungen, auf die über Web-Server zugegriffen wird, und die über eine standardisierte Schnittstelle (öffentlich) verfügbar sind. Die *Web-Services*-Technologie ermöglicht es u. a.,

→*Enterprise Java Beans* (EJB) für Anwendungen verfügbar zu machen, die nicht in Java entwickelt wurden. Durch den Einsatz der *Web-Services*-Technologie steht dem Anwender via →*SOAP* eine einfache, unabhängige, →*XML*-basierte, standardisierte Schnittstelle zu anderen Systemen zur Verfügung.

Wrapper kapseln Datenquellen während der →*Mediator* die einzelnen gekapselten Datenquellen integriert und Benutzeranfragen beantwortet.

XML (Abkürzung für engl. *eXtensible Markup Language*) *XML* ist eine Methode zur Repräsentation strukturierter Daten. *XML* ist (wie auch →*HTML*) eine „vereinfachte" Version der *Standard Generalized Markup Language* (SGML), die es Programmierern von Web-Seiten erleichtert SGML-Anwendungen zu schreiben, und dabei eigene Dokumententypen (DTD) festzulegen.

zlib/libpng-Lizenz Die *zlib/libpng*-Lizenz ist wie die →*GPL* eine Lizenz für freie Software.[6] Sie wurde ursprünglich von den Entwicklern des Dateikompressionsprogramms „gzip" für deren „zlib"-Kompressionsbibliothek genutzt. Selbige ist außerdem integraler Bestandteil der „libpng"-Grafikbibliothek. Die *zlib/libpng*-Lizenz kennt kaum Einschränkungen und ist daher „freier" als z. B. die *GPL*-Lizenz. Sie erlaubt explizit auch die Nutzung in kommerziellen Anwendungen.

[6] `http://www.gzip.org/zlib/zlib_license.html`

Literaturverzeichnis

Hinweis: Die Zahlen am Ende eines jeden Eintrags kennzeichnen die Seitenzahlen der vorliegenden Arbeit, auf denen die jeweilige Quelle zitiert wird.

1. **Abdennadher et al. 2001** ABDENNADHER, Slim ; KRÄMER, Ekkerhard ; SAFT, Matthias ; SCHMAUSS, Matthias: JACK: A Java Constraint Kit. In: *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP'01)*, Universität Kiel (veröffentlicht als technischer Bericht Nr. 2017), 13.–15. September 2001. – URL `http://www.pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2001-10.pdf`. – Zugriffsdatum: 16. März 2005. – Zugl.: (Abdennadher et al. 2002a). – Gekürzte Fassung: (Abdennadher et al. 2002b) 76, 337

2. **Abdennadher et al. 2002a** ABDENNADHER, Slim ; KRÄMER, Ekkerhard ; SAFT, Matthias ; SCHMAUSS, Matthias: JACK: A Java Constraint Kit. In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 64 (2002), September, S. 1–17. – URL `http://www.cs.guc.edu.eg/faculty/sabdennadher/Publikationen/paperENTC.ps.gz`. – Zugriffsdatum: 16. März 2005. – Zugl.: (Abdennadher et al. 2001). – Gekürzte Fassung: (Abdennadher et al. 2002b). – ISSN 1571-0661 76, 337

3. **Abdennadher et al. 2002b** ABDENNADHER, Slim ; KRÄMER, Ekkerhard ; SAFT, Matthias ; SCHMAUSS, Matthias: JACK: A Java Constraint Kit. In: *ALP Newsletter (Association for Logic Programming)* 15 (2002), Februar, Nr. 1. – URL `http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/feb02/nav/monfroy/monfroy_toprint.pdf`. – Zugriffsdatum: 2. Februar 2005. – Ausführliche Version: (Abdennadher et al. 2001, 2002a) 76, 337

4. **Alefeld und Herzberger 1974** ALEFELD, Götz ; HERZBERGER, Jürgen: *Einführung in die Intervallrechnung*. Mannheim, Wien, Zürich : Bibliographisches Institut, Wissenschaftsverlag, 1974 (Reihe Informatik 12). – xiii + 398 S. – ISBN 3-411-01466-0 146, 147, 148

5. **Allen 1983** ALLEN, James F.: Maintaining Knowledge about Temporal Intervals. In: *Communications of the ACM (CACM)* 26 (1983), November, Nr. 11, S. 832–843. – ISSN 0001-0782 58

6. **Arbab 1998a** Arbab, Farhad: The Coordination Language Manifold. In: *ERCIM News, online edition* (1998), Oktober, Nr. 35. – URL http://www.ercim.org/publication/Ercim_News/enw35/arbab.html. – Zugriffsdatum: 6. Juni 2005 78

7. **Arbab 1998b** Arbab, Farhad: What Do You Mean, Coordination? In: Brune, Mieké (Hrsg.) ; Willem Klop, Jan (Hrsg.) ; Rutten, Jan (Hrsg.): *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*. Amsterdam, The Netherlands, März 1998, S. 11–22. – URL http://www.cwi.nl/pub/manifold/NVTIpaper.ps.Z. – Zugriffsdatum: 31. Mai 2005. – March'98 Issue of the Bulletin of the Dutch Association for Theoretical Computer Science 78

8. **Arbab und Monfroy 1998a** Arbab, Farhad ; Monfroy, Eric: Coordination of Heterogeneous Distributed Cooperative Constraint Solving. In: *ACM SIGAPP Applied Computing Review* 6 (1998), September, Nr. 2, S. 4–17. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/monfroy/Papers/acr98.ps.gz. – Zugriffsdatum: 1. Juni 2005. – Special Issue on Coodination Languages and Models. – Zugl.: CWI Technical Report SEN-R9828, ISSN 1386-369X. – Vorhergehende Version: (Arbab und Monfroy 1998b) 78, 338

9. **Arbab und Monfroy 1998b** Arbab, Farhad ; Monfroy, Eric: Using Coordination for Cooperative Constraint Solving. In: George, K. M. (Hrsg.) ; Lamong, Gary B. (Hrsg.): *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC'98), Atlanta, Georgia, USA, 27. Februar – 1. März 1998*. New York, NY, USA : ACM Press, 1998, S. 139–148. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/monfroy/Papers/sac98.ps.gz. – Zugriffsdatum: 1. Juni 2005. – Erweiterte Fassung: (Arbab und Monfroy 1998a). – ISBN 0-89791-969-6 78, 338

10. **Ardissono et al. 2001** Ardissono, Liliana ; Felfernig, Alexander ; Friedrich, Gerhard ; Goy, Anna ; Jannach, Dietmar ; Schäfer, Ralph ; Zanker, Markus: Web-Based Commerce of Complex Products and Services with Multiple Suppliers. In: Sauer, Jürgen (Hrsg.): *Proceedings of the 15th Workshop "AI in Planning, Scheduling, Configuration and Design" (PuK'01) at the 24th Joint German/Austrian Conference on Artificial Intelligence (KI'01)*. Wien, Österreich, 18. September 2001, S. 1–8. – URL http://www-is.informatik.uni-oldenburg.de/~sauer/puk2001/papers/zanker.pdf. – Zugriffsdatum: 10. Oktober 2004 270

11. **Ardissono et al. 2002** Ardissono, Liliana ; Felfernig, Alexander ; Friedrich, Gerhard ; Jannach, Dietmar ; Schäfer, Ralph ; Zanker, Markus: A Framework for Rapid Development of Advanced Web-Based Configurator Applications. In: Harmelen, Frank van (Hrsg.): *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'02) – Prestigious Applications of Intelligent Systems (PAIS), Lyon, France, 21.–26. Juli 2002*. Amsterdam, The Netherlands : IOS Press, September 2002 (Frontiers in Artificial Intelligence and Applications 77), S. 618–622. – URL http://www.di.unito.it/~liliana/EC/pais02.pdf. – Zugriffsdatum: 15. September 2004. – ISBN 1-586-03257-7 18, 270, 271

12. **Arlt et al. 1999** ARLT, Volker ; GÜNTER, Andreas ; HOLLMANN, Oliver ; WAGNER, Thomas ; HOTZ, Lothar: EngCon – Engineering & Configuration. In: (Faltings et al. 1999), S. 123–124. – ISBN 1-57735-089-8 21

13. **Armstrong et al. 2005** ARMSTRONG, Eric ; BALL, Jennifer ; BODOFF, Stephanie ; BODE CARSON, Debbie ; EVANS, Ian ; GREEN, Dale ; HAASE, Kim ; JENDROCK, Eric: The J2EE 1.4 Tutorial – For Sun Java System Application Server Platform / Sun Microsystems. Santa Clara, California, USA, 7. Juni 2005 (Edition 8.1 2005Q2 UR2). – Tutorial. – xlii + 1499 S. – URL `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/J2EETutorial.pdf`. – Zugriffsdatum: 7. November 2005 200

14. **Axling und Haridi 1996** AXLING, Tomas ; HARIDI, Seif: A Tool for Developing Interactive Configuration Applications. In: *The Journal of Logic Programming* 26 (1996), Februar, Nr. 2, S. 147–168. – URL `http://www.sics.se/~axling/obelics.ps.Z`. – Zugriffsdatum: 13. Oktober 2004. – ISSN 0743-1066 267

15. **Bacchus 2000** BACCHUS, Fahiem: Extending Forward Checking. In: (Dechter 2000), S. 35–51. – URL `http://www.cs.toronto.edu/~fbacchus/Papers/efc.pdf`. – Zugriffsdatum: 21. September 2004. – ISBN 3-540-41053-8 122

16. **Bacchus und van Beek 1998** BACCHUS, Fahiem ; BEEK, Peter van: On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems. In: RICH, Charles (Hrsg.) ; MOSTOW, Jack (Hrsg.) ; BUCHANAN, Bruce G. (Hrsg.) ; UTHURUSAMY, Ramasamy (Hrsg.): *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98) and 10th Conference on Innovative Applications of Artificial Intelligence (IAAI'98), Madison, Wisconsin, USA, 26.–30. Juli 1998*. Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1998, S. 311–318. – URL `http://www.cs.toronto.edu/~fbacchus/Papers/BvBAAAI98.pdf`. – Zugriffsdatum: 21. September 2004. – ISBN 0-262-51098-7 138, 140, 141, 142

17. **Bacchus et al. 2002** BACCHUS, Fahiem ; CHEN, Xinguang ; BEEK, Peter van ; WALSH, Toby: Binary vs. Non-Binary Constraints. In: *Artificial Intelligence* 140 (2002), September, Nr. 1–2, S. 1–37. – URL `http://www.cs.toronto.edu/~fbacchus/Papers/bcvwaij02.pdf`. – Zugriffsdatum: 21. September 2004. – ISSN 0004-3702 140, 141

18. **Bacchus und Grove 1995** BACCHUS, Fahiem ; GROVE, Adam J.: On the Forward Checking Algorithm. In: (Montanari und Rossi 1995), S. 292–309. – URL `http://www.cs.toronto.edu/~fbacchus/Papers/BGCP95.pdf`. – Zugriffsdatum: 21. September 2004. – ISBN 3-540-60299-2 129, 132

19. **Bacchus und van Run 1995** BACCHUS, Fahiem ; RUN, Paul van: Dynamic Variable Ordering in CSPs. In: (Montanari und Rossi 1995), S. 258–275. – URL `http://www.cs.toronto.edu/~fbacchus/Papers/BvRCP95.pdf`. – Zugriffsdatum: 21. September 2004. – ISBN 3-540-60299-2 129, 132

20. **Badros et al. 2001** Badros, Greg J. ; Borning, Alan ; Stuckey, Peter J.: The Cassowary Linear Arithmetic Constraint Solving Algorithm. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 8 (2001), Nr. 4, S. 267–306. – URL `http://www.cs.washington.edu/research/constraints/solvers/cassowary-tochi.pdf`. – Zugriffsdatum: 7. Februar 2005. – ISSN 1073-0516 69

21. **Barták 1999a** Barták, Roman: Constraint Programming: In Pursuit of the Holy Grail. In: *Proceedings of the Week of Doctoral Students (WDS'99), Part IV (Invited Lecture)*. Prague, Czechia : MatFyzPress, Juni 1999, S. 555–564. – URL `http://ktilinux.ms.mff.cuni.cz/~bartak/downloads/WDS99.pdf`. – Zugriffsdatum: 19. Januar 2006. – Vorhergehende Version: (Barták 1999b) 83, 89, 90, 104, 113, 116, 123, 340

22. **Barták 1999b** Barták, Roman: Constraint Programming: What is Behind? In: *Proceedings of the Workshop on Constraint Programming for Decision and Control (CPDC'99), Invited Talk*. Gliwice, Poland, Juni 1999, S. 7–15. – URL `http://ktilinux.ms.mff.cuni.cz/~bartak/downloads/CPDC99.pdf`. – Zugriffsdatum: 22. September 2004. – Erweiterte Fassung: (Barták 1999a) 52, 83, 89, 90, 104, 113, 340

23. **Barták 2001** Barták, Roman: Theory and Practice of Constraint Propagation. In: *Proceedings of the 3rd Workshop on Constraint Programming for Decision and Control (CPDC'01), Invited Lecture*. Gliwice, Poland, Juni 2001, S. 7–14. – URL `http://ktilinux.ms.mff.cuni.cz/~bartak/downloads/CPDC2001.zip`. – Zugriffsdatum: 19. Januar 2006 53, 83, 84, 103, 108, 135, 138

24. **Barták 2003** Barták, Roman: On-Line Guide to Constraint Programming / Charles University, Department of Theoretical Computer Science and Mathematical Logic, Faculty of Mathematics and Physics. Czechia, Prague, Stand: 4. Oktober 2003. – Tutorial. – URL `http://kti.ms.mff.cuni.cz/~bartak/constraints`. – Zugriffsdatum: 8. April 2004 128, 134, 139, 140, 141

25. **Bauch et al. 1987** Bauch, Hartmut ; Jahn, Karl-Udo ; Oelschlägel, Dieter ; Süsse, Herbert ; Wiebigke, Volkmar: *Intervallmathematik, Theorie und Anwendungen*. 1. Aufl. Leipzig : BSB B. G. Teubner Verlagsgesellschaft, 1987 (Mathematisch-Naturwissenschaftliche Bibliothek 72). – 260 S. – ISBN 3-322-00384-1 145, 146, 147, 148, 149

26. **van Beek 1992** Beek, Peter van: On the Minimality and Decomposability of Constraint Networks. In: Rosenbloom, Paul (Hrsg.) ; Szolovits, Peter (Hrsg.): *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92), San Jose, California, USA, 12.–16. Juli 1992*. Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1992, S. 447–452. – URL `http://ai.uwaterloo.ca/~vanbeek/publications/aaai92.ps.gz`. – Zugriffsdatum: 15. September 2004. – ISBN 0-262-51063-4 103

27. **Benhamou 1995** Benhamou, Frédéric: Interval Constraint Logic Programming. In: (Podelski 1995), S. 1–21. – URL `http://www.sciences.univ-nantes.fr/info/`

perso/permanents/benhamou/PAPERS/Ben_Chatillon95.pdf. – Zugriffsdatum: 27. Dezember 2004. – ISBN 3-540-59155-9 83, 159, 162

28. **Benhamou 1996** BENHAMOU, Frédéric: Heterogeneous Constraint Solving. In: HANUS, Michael (Hrsg.) ; RODRÍGUEZ-ARTALEJO, Mario (Hrsg.): *Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96), Aachen, 25.–27. September 1996*. Berlin, Heidelberg, New York : Springer Verlag, 1996 (LNCS 1139), S. 62–76. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/Ben_ALP96.pdf. – Zugriffsdatum: 15. Mai 2005. – ISBN 3-540-61735-3 65, 77, 190

29. **Benhamou 2001** BENHAMOU, Frédéric: Interval Constraints. In: FLOUDAS, Christodoulos A. (Hrsg.) ; PARDALOS, Panos M. (Hrsg.): *Encyclopedia of Optimization* Bd. 3. Dordrecht, Netherlands : Kluwer Academic Publishers, August 2001, S. 45–48. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/Ben99_EoO.pdf. – Zugriffsdatum: 14. September 2004. – ISBN 0-7923-6932-7 143, 144, 159

30. **Benhamou 2002** BENHAMOU, Frédéric: *Interval Reasoning*. Invited Tutorial (Slides) at the 8th International Conference on Principles and Practice of Constraint Programming (CP'02), Ithaca, New York, USA. 13. September 2002. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/Ben_TutorialCP2002.pdf. – Zugriffsdatum: 7. Oktober 2004 166

31. **Benhamou et al. 1999a** BENHAMOU, Frédéric ; GOUALARD, Frédéric ; GRANVILLIERS, Laurent ; PUGET, Jean-François: Revising Hull and Box Consistency. In: DE SCHREYE, Danny (Hrsg.): *Proceedings of the 16th International Conference on Logic Programming (ICLP'99), Las Cruces, New Mexico, USA, 29. November – 4. Dezember 1999*. Cambridge, Massachusetts, USA : The MIT Press, 1999, S. 230–244. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/BenGouGranPug_ICLP99.pdf. – Zugriffsdatum: 14. September 2004. – ISBN 0-262-54104-1 159, 162, 167

32. **Benhamou et al. 1999b** BENHAMOU, Frédéric ; GOUALARD, Frédéric ; GRANVILLIERS, Laurent ; PUGET, Jean-François: *Revising Hull and Box Consistency*. Lecture Slides at the 16th International Conference on Logic Programming (ICLP'99), Las Cruces, New Mexico, USA. 30. November 1999. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/Slides_ICLP99.pdf. – Zugriffsdatum: 7. Oktober 2004 165

33. **Benhamou und Granvilliers 1996** BENHAMOU, Frédéric ; GRANVILLIERS, Laurent: Combining Local Consistency, Symbolic Rewriting and Interval Methods. In: CALMET, Jacques (Hrsg.) ; CAMPBELL, John A. (Hrsg.) ; PFALZGRAF, Jochen (Hrsg.): *Proceedings of the 3rd International Conference on Artificial Intelligence and Symbolic Mathematical Computation (AISMC-3), Steyr, Austria, 23.–25. September 1996*. Berlin, Heidelberg, New York : Springer Verlag, 1996 (LNCS

1138), S. 144–159. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/granvil/papers/bgaismc96.pdf. – Zugriffsdatum: 9. Juni 2005. – ISBN 3-540-61732-9 77

34. **Benhamou et al. 1994a** Benhamou, Frédéric ; McAllester, David ; Van Hentenryck, Pascal: CLP(Intervals) Revisited. In: Bruynooghe, Maurice (Hrsg.): *Logic Programming, Proceedings of the 1994 International Symposium (ILPS'94), Ithaca, New York, USA, 13.–17. November 1994.* Cambridge, Massachusetts, USA : The MIT Press, 1994, S. 124–138. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/BenMcAlVHen94.pdf. – Zugriffsdatum: 14. September 2004. – Vorhergehende Version: (Benhamou et al. 1994b). – ISBN 0-262-52191-1 145, 162, 163, 165, 173, 342

35. **Benhamou et al. 1994b** Benhamou, Frédéric ; McAllester, David ; Van Hentenryck, Pascal: CLP(Intervals) Revisited / University of Marseille. France, April 1994 (Technical Report CS-94-18). – Forschungsbericht. – 16 S. – URL ftp://ftp.cs.brown.edu/pub/techreports/94/cs94-18.ps.Z. – Zugriffsdatum: 22. September 2004. Überarb. Fassung: (Benhamou et al. 1994a) 162, 342

36. **Benhamou und Older 1997** Benhamou, Frédéric ; Older, William: Applying Interval Arithmetic to Real, Integer and Boolean Constraints. In: *The Journal of Logic Programming* 32 (1997), Juli, Nr. 1, S. 1–24. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/BenOld_JLP97.pdf. – Zugriffsdatum: 14. September 2004. – ISSN 0743-1066 162

37. **Benhamou und Van Hentenryck 1997** Benhamou, Frédéric ; Van Hentenryck, Pascal: Introduction to the Special Issue on Interval Constraints. In: *Constraints, An International Journal* 2 (1997), April, Nr. 2, S. 107–112. – ISSN 1383-7133 67

38. **Berk 2000** Berk, Elliot: JLex: A lexical analyzer generator for Java / Princeton University, Department of Computer Science. Princeton, New Jersey, USA, 6. September 2000 (Version 1.2.5). – Online Handbuch. – URL http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html. – Zugriffsdatum: 14. September 2005 200, 204, 281

39. **Berlandier 1995** Berlandier, Pierre: Improving Domain Filtering Using Restricted Path Consistency. In: *Proceedings of the 11th Conference on Artificial Intelligence for Applications (CAIA'95), Los Angeles, California, USA, 20.–22. Februar 1995.* Los Alamitos, California, USA, : IEEE Computer Society Press, 1995, S. 32–37. – URL ftp://ftp-sop.inria.fr/coprin/neveu/contraintes/berlandier/caia95.ps.gz. – Zugriffsdatum: 15. September 2004. – ISBN 0-8186-7070-3 109

40. **Bessière 1994a** Bessière, Christian: Arc-Consistency and Arc-Consistency Again. In: *Artificial Intelligence* 65 (1994), Januar, Nr. 1, S. 179–190. – URL

http://www.lirmm.fr/~bessiere/stock/aij94.pdf. – Zugriffsdatum: 30. März 2005. – Vorhergehende Version: (Bessière und Cordier 1993). – ISSN 0004-3702 96, 106, 122, 343

41. **Bessière 1994b** Bessière, Christian: A Fast Algorithm to Establish Arc-Consistency in Constraint Networks / LIRMM, Université de Montpellier II. Montpellier, France, Januar 1994 (Technical Report TR-94-003). – Forschungsbericht. – 6 S. – URL http://citeseer.ist.psu.edu/bessi94fast.html. – Zugriffsdatum: 16. September 2004 96

42. **Bessière 1999** Bessière, Christian: Non-Binary Constraints (Invited Paper). In: (Jaffar 1999), S. 24–27. – URL http://www.lirmm.fr/~bessiere/stock/cp99-nonbinary.ps. – Zugriffsdatum: 17. September 2004. – ISBN 3-540-66626-5 135

43. **Bessière und Cordier 1993** Bessière, Christian ; Cordier, Marie-Odile: Arc-Consistency and Arc-Consistency Again. In: Fikes, Richard (Hrsg.) ; Lehnert, Wendy (Hrsg.): *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'93), Washington, DC, USA, 11.–15. Juli 1993.* Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1993, S. 108–113. – URL http://www.lirmm.fr/~bessiere/stock/aaai93.pdf. – Zugriffsdatum: 30. März 2005. – Überarb. Fassung: (Bessière 1994a). – ISBN 0-262-51071-5 96, 106, 343

44. **Bessière et al. 1995** Bessière, Christian ; Freuder, Eugene C. ; Régin, Jean-Charles: Using Inference to Reduce Arc Consistency Computation. In: Perrault, C. Raymond (Hrsg.) ; Mellish, Chris S. (Hrsg.): *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95), Montréal, Québec, Canada, 20.–25. August 1995* Bd. 1. San Mateo, California, USA : Morgan Kaufmann Publishers, Dezember 1995, S. 592–599. – URL http://www.lirmm.fr/~bessiere/stock/ijcai95.ps. – Zugriffsdatum: 15. September 2004. – ISBN 1-558-60363-8 97, 106

45. **Bessière et al. 1999a** Bessière, Christian ; Freuder, Eugene C. ; Régin, Jean-Charles: Using Constraint Metaknowledge to Reduce Arc Consistency Computation. In: *Artificial Intelligence* 107 (1999), Januar, Nr. 1, S. 125–148. – URL http://www.lirmm.fr/~bessiere/stock/aij99.ps. – Zugriffsdatum: 21. September 2004. – ISSN 0004-3702 97, 106

46. **Bessière et al. 1999b** Bessière, Christian ; Meseguer, Pedro ; Freuder, Eugene C. ; Larrosa, Javier: On Forward Checking for Non-Binary Constraint Satisfaction. In: Régin, Jean-Charles (Hrsg.) ; Nuijten, Wim (Hrsg.): *Proceedings of the Workshop on Non-Binary Constraints at the 16th International Joint Conference on Artificial Intelligence (IJCAI'99).* Stockholm, Sweden, 2. August 1999, S. 9–16. – URL http://www.lirmm.fr/~bessiere/stock/ijcai99ws.ps. – Zugriffsdatum: 16. September 2004. – Zugl.: (Bessière et al. 1999c). – Erweiterte Fassung: (Bessière et al. 2002) 136, 344

47. **Bessière et al. 1999c** Bessière, Christian ; Meseguer, Pedro ; Freuder, Eugene C. ; Larrosa, Javier: On Forward Checking for Non-Binary Constraint Satisfaction. In: (Jaffar 1999), S. 88–102. – URL http://www.lirmm.fr/~bessiere/stock/cp99-nfc.ps. – Zugriffsdatum: 14. September 2004. – Zugl.: (Bessière et al. 1999b). – Erweiterte Fassung: (Bessière et al. 2002). – ISBN 3-540-66626-5 136, 343, 344

48. **Bessière et al. 2002** Bessière, Christian ; Meseguer, Pedro ; Freuder, Eugene C. ; Larrosa, Javier: On Forward Checking for Non-Binary Constraint Satisfaction. In: *Artificial Intelligence* 141 (2002), Oktober, Nr. 1–2, S. 205–224. – URL http://www.lirmm.fr/~bessiere/stock/aij02.ps. – Zugriffsdatum: 16. September 2004. – Vorhergehende Versionen: (Bessière et al. 1999b, c). – ISSN 0004-3702 136, 343, 344

49. **Bessière und Régin 1995** Bessière, Christian ; Régin, Jean-Charles: Using Bidirectionality to Speed Up Arc-Consistency Processing. In: Meyer, Manfred (Hrsg.): *Constraint Processing, Selected Papers*. Berlin, Heidelberg, New York : Springer Verlag, 1995 (LNCS 923), S. 157–169. – URL http://www.lirmm.fr/~bessiere/stock/lncs923.ps. – Zugriffsdatum: 16. September 2004. – ISBN 3-540-59479-5 96, 97

50. **Bessière und Régin 1996** Bessière, Christian ; Régin, Jean-Charles: MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In: (Freuder 1996), S. 61–75. – URL http://www.lirmm.fr/~bessiere/stock/cp96.ps. – Zugriffsdatum: 14. September 2004. – ISBN 3-540-61551-2 118, 125, 127, 129, 130, 131, 132, 133, 134, 210, 312

51. **Bessière und Régin 1997** Bessière, Christian ; Régin, Jean-Charles: Arc Consistency for General Constraint Networks: Preliminary Results. In: (Pollack und Georgeff 1997), S. 398–404. – URL http://www.lirmm.fr/~bessiere/stock/ijcai97-gac.ps. – Zugriffsdatum: 15. September 2004. – ISBN 1-558-60480-4 137

52. **Bessière und Régin 1998** Bessière, Christian ; Régin, Jean-Charles: Local Consistency on Conjunctions of Constraints. In: Régin, Jean-Charles (Hrsg.) ; Nuijten, Wim (Hrsg.): *Proceedings of the Workshop on Non-Binary Constraints at the 13th Biennial European Conference on Artificial Intelligence (ECAI'98)*. Brighton, UK, 23.–28. August 1998, S. 53–59. – URL http://www.lirmm.fr/~bessiere/stock/ecai98ws.ps. – Zugriffsdatum: 18. September 2004 137

53. **Bessière und Régin 2001a** Bessière, Christian ; Régin, Jean-Charles: Refining the Basic Constraint Propagation Algorithm. In: (Nebel 2001), S. 309–315. – URL http://www.lirmm.fr/~bessiere/stock/ijcai01.ps. – Zugriffsdatum: 20. September 2004. – Zugl.: (Bessière und Régin 2001b). – ISBN 1-55860-777-3 97, 106, 345

54. **Bessière und Régin 2001b** Bessière, Christian ; Régin, Jean-Charles: Refining the Basic Constraint Propagation Algorithm. In: Codognet, Philippe

(Hrsg.): *Proceedings of the 10th International French Speaking Conference on Logic and Constraint Programming: Actes des Dixièmes Journées Francophones de Programmation en Logique et Programmation par Contraintes (JFPLC'01), Paris, France, 24.–27. April 2001.* Paris : Hermès Science Publications, 2001. – URL `http://www.lirmm.fr/~bessiere/stock/jfplc01.ps`. – Zugriffsdatum: 20. September 2004. – Zugl.: (Bessière und Régin 2001a). – ISBN 2-7462-0255-7 97, 344

55. **Bitner und Reingold 1975** Bitner, James R. ; Reingold, Edward M.: Backtrack Programming Techniques. In: *Communications of the ACM (CACM)* 18 (1975), November, Nr. 11, S. 651–656. – ISSN 0001-0782 114, 129

56. **Bonar 2003** Bonar, Pascal: *Unterstützung der Modellierung von Abläufen in strukturbasierten Konfigurationssystemen*, Universität Bremen, Diplomarbeit, 18. Dezember 2003. – iv + 210 S. 21, 23

57. **Bordeaux et al. 2001** Bordeaux, Lucas ; Monfroy, Eric ; Benhamou, Frédéric: Improved Bounds on the Complexity of kB-Consistency. In: (Nebel 2001), S. 303–308. – URL `http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/BorMonBen_IJCAI2001.pdf`. – Zugriffsdatum: 20. September 2004. – ISBN 1-55860-777-3 161

58. **Borning et al. 1987** Borning, Alan ; Duisberg, Robert ; Freeman-Benson, Bjørn N. ; Kramer, Axel ; Woolf, Michael: Constraint Hierarchies. In: Meyrowitz, Norman (Hrsg.): *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Orlando, Florida, USA*. New York, NY, USA : ACM Press, Dezember 1987, S. 48–60. – Zugl.: ACM SIGPLAN Notices, 22 (1987), Nr. 12, S. 48–60. – ISBN 0-89791-247-0 61

59. **Borning et al. 1992** Borning, Alan ; Freeman-Benson, Bjørn N. ; Wilson, Molly: Constraint Hierarchies. In: *Lisp and Symbolic Computation* 5 (1992), September, Nr. 3, S. 223–270. – URL `ftp://ftp.cs.washington.edu/pub/constraints/papers/constraint-hierarchies-lisp-symb-comp.pdf`. – Zugriffsdatum: 27. Oktober 2004. – Zugl.: (Borning et al. 1994) und (Jampel et al. 1996), S. 23–62. – ISSN 0892-4635 61, 62, 345

60. **Borning et al. 1994** Borning, Alan ; Freeman-Benson, Bjørn N. ; Wilson, Molly: Constraint Hierarchies. In: Mayoh, Brian (Hrsg.) ; Tyugu, Enn (Hrsg.) ; Penjam, Jaan (Hrsg.): *Constraint Programming – Proceedings of the NATO Advanced Study Institute on Constraint Programming, Pärnu, Estonia, 13.–24. August 1993*. Berlin, Heidelberg, New York : Springer Verlag, September 1994 (NATO ASI Series F: Computer and System Sciences Vol. 131), Kap. 2.2, S. 75–115. – URL `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/txt/proc/natobook.tgz`. – Zugriffsdatum: 3. März 2005. – Zugl.: (Borning et al. 1992) und (Jampel et al. 1996), S. 23–62. – ISBN 3-540-57859-5 61, 345

61. **Borrett et al. 1995** Borrett, James E. ; Tsang, Edward P. K. ; Walsh, Natasha R.: Adaptive Constraint Satisfaction: The Quickest First Principle / University of Essex, Department of Computer Science. Wivenhoe Park, Colchester, UK, 13. November 1995 (Technical Report CSM-256). – Forschungsbericht. – 25 S. – URL http://cswww.essex.ac.uk/CSP/papers/CSM-256.ps.Z. – Zugriffsdatum: 9. November 2004. Gekürzte Fassung: (Borrett et al. 1996b) 64, 346

62. **Borrett et al. 1996a** Borrett, James E. ; Tsang, Edward P. K. ; Walsh, Natasha R.: Adaptive Constraint Satisfaction. In: *Proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG'96).* University of Liverpool, UK, 21.–22. November 1996. – URL http://cswww.essex.ac.uk/CSP/papers/BoTsWa-Acs-planningsig96.pdf. – Zugriffsdatum: 9. November 2004 64

63. **Borrett et al. 1996b** Borrett, James E. ; Tsang, Edward P. K. ; Walsh, Natasha R.: Adaptive Constraint Satisfaction: The Quickest First Principle. In: Wahlster, Wolfgang (Hrsg.): *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI'96), Budapest, Hungary, 11.–16. August 1996.* Chichester, London, New York : John Wiley & Sons, 1996, S. 160–164. – Ausführliche Version: (Borrett et al. 1995). – ISBN 0-471-96809-9 64, 346

64. **Bronstein et al. 1996** Bronstein, Ilja N. ; Semendjajew, Konstantin A. ; Grosche, Günter ; Ziegler, Viktor ; Ziegler, Dorothea ; Zeidler, Eberhard (Hrsg.): *Teubner-Taschenbuch der Mathematik.* Bd. I. 1. Aufl. XXVI. Stuttgart, Leipzig : B. G. Teubner Verlagsgesellschaft, Mai 1996. – xxv + 1298 S. – ISBN 3-8154-2001-6 83, 146, 147, 164

65. **Broy 1992** Broy, Manfred: *Informatik, Eine grundlegende Einführung.* Bd. I. Berlin, Heidelberg, New York : Springer Verlag, 1992. – xii + 250 S. – ISBN 3-540-55191-3 10, 50

66. **Broy 1995** Broy, Manfred: *Informatik, Eine grundlegende Einführung.* Bd. IV. Berlin, Heidelberg, New York : Springer Verlag, 1995. – ix + 215 S. – ISBN 3-540-58602-4 84

67. **Campus Press 2002** Technologie-Zentrum Informatik gewinnt internationalen Innovationspreis / Campus Press. Pressestelle der Universität Bremen, 14. Juni 2002. – Pressemitteilung Nr. 135. – URL http://www.uni-bremen.de/campus/campuspress/altpress/02-135.php3. – Zugriffsdatum: 8. November 2004 21

68. **Cheeseman et al. 1991** Cheeseman, Peter ; Kanefsky, Bob ; Taylor, William M.: Where the *Really* Hard Problems Are. In: Mylopoulos, John (Hrsg.) ; Reiter, Raymond (Hrsg.): *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91), Sydney, Australia, 24.–30. August 1991* Bd. 1. San Mateo, California, USA : Morgan Kaufmann Publishers, Dezember 1991, S. 331–337. – URL http://ic-www.arc.nasa.gov/ic/projects/bayes-group/NP/ijcai91/phases-for-web.ps. – Zugriffsdatum: 12. Oktober 2004. – ISBN 1-55860-160-0 56, 57

69. **Chen und van Beek 2001** CHEN, Xinguang ; BEEK, Peter van: Conflict-Directed Backjumping Revisited. In: *Journal of Artificial Intelligence Research (JAIR)* 14 (2001), S. 53–81. – URL `http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume14/chen01a.pdf`. – Zugriffsdatum: 21. September 2004 125, 126, 127, 129, 131

70. **Chmeiss 1996** CHMEISS, Assef: Sur la Consistance de Chemin et ses Formes partielles. In: *Actes du Congrès de l'Association Française pour la Cybernétique Economique et Technique de Reconnaissance des Formes et Intelligence Artificielle (AFCET-RFIA'96).* Rennes, France, 16.–18. Januar 1996, S. 212–219 102, 106

71. **Chmeiss und Jégou 1995** CHMEISS, Assef ; JÉGOU, Philippe: Partial and Global Path Consistency Revisited / Laboratoire d'Informatique de Marseille. France, 1995 (Technical Report 120.95). – Forschungsbericht 102, 106

72. **Chmeiss und Jégou 1996a** CHMEISS, Assef ; JÉGOU, Philippe: Path-Consistency: When Space Misses Time. In: (Clancey et al. 1996), S. 196–201. – ISBN 0-262-51091-X 102, 106

73. **Chmeiss und Jégou 1996b** CHMEISS, Assef ; JÉGOU, Philippe: Two New Constraint Propagation Algorithms Requiring Small Space Complexity. In: RADLE, Mark G. (Hrsg.): *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'96), Toulousse, France, 16.–19. November 1996.* Los Alamitos, California, USA, : IEEE Computer Society Press, 1996, S. 286–289. – ISBN 0-8186-7686-8 97, 102, 106

74. **Chmeiss und Jégou 1998** CHMEISS, Assef ; JÉGOU, Philippe: Efficient Path-Consistency Propagation. In: *International Journal on Artificial Intelligence Tools (IJAIT)* 7 (1998), März, Nr. 2, S. 121–142. – ISSN 0218-2130 85, 97, 102, 106

75. **Chopra 1997** CHOPRA, Rajiv: *An Architecture for Exploiting Qualitative Scene-Specific Context in High-Level Computer Vision.* Buffalo, New York, USA, The State University of New York at Buffalo, Center of Excellence for Document Analysis and Recognition (CEDAR), PhD Thesis, Juni 1997. – URL `http://www.cedar.buffalo.edu/~rchopra/diss.ps.gz`. – Zugriffsdatum: 21. September 2004. – ix + 120 S. 143

76. **Chun 1999a** CHUN, Andy Hon Wai: Constraint Programming in Java with JSolver. In: *Proceedings of the 1st International Conference and Exhibition on Practical Application of Constraint Technologies and Logic Programming (PACLP'99), London, UK, 19.–21. April 1999.* Blackpool, Lancashire, UK : The Practical Applications Company, 1999. – URL `http://www.cs.cityu.edu.hk/~hwchun/research/PDF/PACLP99JSolver.pdf`. – Zugriffsdatum: 15. Februar 2005 69

77. **Chun 1999b** CHUN, Andy Hon Wai: Constraint Programming in Java with JSolver 2.0 – An Introduction / City University of Hong Kong, Department of Electronic Engineering. Kowloon, Hong Kong, 15. November 1999. – Forschungsbericht. –

URL http://www.e-optimization.com/resources/uploads/JSolver_2.0_Intro.pdf. – Zugriffsdatum: 15. Februar 2005 69

78. **Chun 1999c** CHUN, Andy Hon Wai: Waltz Filtering in Java with JSolver. In: *Proceedings of the 1st International Conference and Exhibition on Practical Application of Java (PA Java'99), London, UK, 21.–23. April 1999*. Blackpool, Lancashire, UK : The Practical Applications Company, 1999. – URL http://www.cs.cityu.edu.hk/~hwchun/research/PDF/PAJava99Waltz.pdf. – Zugriffsdatum: 15. Februar 2005 69

79. **Clancey et al. 1996** CLANCEY, William J. (Hrsg.) ; WELD, Dan (Hrsg.) ; SHROBE, Howard E. (Hrsg.) ; SENATOR, Ted E. (Hrsg.): *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96) and 8th Conference on Innovative Applications of Artificial Intelligence (IAAI'96), Portland, Oregon, USA, 4.–8. August 1996*. Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1996. – ISBN 0-262-51091-X 347, 356, 379

80. **Claus und Schwill 2001** CLAUS, Volker (Hrsg.) ; SCHWILL, Andreas (Hrsg.): *Duden Informatik, Ein Fachlexikon für Studium und Praxis*. 3. Aufl. Mannheim, Leipzig, Wien, Zürich : Dudenverlag, Bibliographisches Institut, 2001. – 762 S. – ISBN 3-411-05233-3 84, 327, 328, 329, 331, 333, 334, 335

81. **Cleary 1987** CLEARY, John G.: Logical Arithmetic. In: *Future Computing Systems* 2 (1987), Nr. 2, S. 125–149. – ISSN 0266-7207 144, 149, 150, 151

82. **Cohen 1990** COHEN, Jacques: Constraint Logic Programming Languages. In: *Communications of the ACM (CACM)* 33 (1990), Juli, Nr. 7, S. 52–68. – ISSN 0001-0782 53

83. **Collavizza et al. 1998** COLLAVIZZA, Hélène ; DELOBEL, François ; RUEHER, Michel: A Note on Partial Consistencies over Continuous Domains. In: MAHER, Michael J. (Hrsg.) ; PUGET, Jean-François (Hrsg.): *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP'98), Pisa, Italy, 26.–30. Oktober 1998*. Berlin, Heidelberg, New York : Springer Verlag, 1998 (LNCS 1520), S. 147–161. – URL http://www.essi.fr/~rueher/Publis/cp98.pdf. – Zugriffsdatum: 14. September 2004. – Erweiterte Fassung: (Collavizza et al. 1999). – ISBN 3-540-21834-3 159, 163, 166, 348

84. **Collavizza et al. 1999** COLLAVIZZA, Hélène ; DELOBEL, François ; RUEHER, Michel: Comparing Partial Consistencies. In: *Reliable Computing* 5 (1999), August, Nr. 3, S. 213–228. – URL http://www.essi.fr/~rueher/Publis/reliable_comp99.ps. – Zugriffsdatum: 14. September 2004. – Vorhergehende Version: (Collavizza et al. 1998). – ISSN 1385-3139 159, 161, 163, 166, 348

85. **Cooper 1989** COOPER, Martin C.: An Optimal k-Consistency Algorithm. In: *Artificial Intelligence* 41 (1989), November, Nr. 1, S. 89–95. – ISSN 0004-3702 105

86. **COPRIN 2004** `ALIAS-C++` – A C++ Algorithms Library of Interval Analysis for equation Systems / The COPRIN Project, INRIA. Sophia Antipolis, France, September 2004 (Version 2.3). – Handbuch. – 190 S. – URL `http://www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS-C++/manual-alias-C++2.3.ps`. – Zugriffsdatum: 11. Februar 2005 70

87. **Cruz und Barahona 1999** Cruz, Jorge ; Barahona, Pedro: An Interval Constraint Approach to Handle Parametric Ordinary Differential Equations for Decision Support. In: (Jaffar 1999), S. 478–479 (Poster). – URL `http://ssdi.di.fct.unl.pt/~jc/CruzCP99.pdf`. – Zugriffsdatum: 14. September 2004. – ISBN 3-540-66626-5 162

88. **Cruz und Barahona 2001** Cruz, Jorge ; Barahona, Pedro: Global Hull Consistency with Local Search for Continuous Constraint Solving. In: Brazdil, Pavel (Hrsg.) ; Jorge, Alípio (Hrsg.): *Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, Proceedings of the 10th Portuguese Conference on Artificial Intelligence (EPIA'01), Porto, Portugal, 17.–20. Dezember 2001.* Berlin, Heidelberg, New York : Springer Verlag, 2001 (LNCS 2258), S. 349–362. – URL `http://ssdi.di.fct.unl.pt/~jc/CruzEpia01.pdf`. – Zugriffsdatum: 17. September 2004. – ISBN 3-540-43030-X 162

89. **Cruz und Barahona 2003** Cruz, Jorge ; Barahona, Pedro: Maintaining Global Hull Consistency with Local Search for Continuous CSPs. In: Bliek, Christian (Hrsg.) ; Jermann, Christophe (Hrsg.) ; Neumaier, Arnold (Hrsg.): *Global Optimization and Constraint Satisfaction, Proceedings of the 1st International Workshop on Global Constraint Optimization and Constraint Satisfaction (COCOS'02), Valbonne-Sophia Antipolis, France, 2.–4. Oktober 2002, Revised Selected Papers.* Berlin, Heidelberg, New York : Springer Verlag, 2003 (LNCS 2861), S. 178–193. – URL `http://ssdi.di.fct.unl.pt/~jc/CruzCocos02_postprocs.pdf`. – Zugriffsdatum: 17. September 2004. – ISBN 3-540-20463-6 143, 162

90. **Cunis 1991** Cunis, Roman: Modellierung technischer Systeme in der Begriffshierarchie. In: (Cunis et al. 1991), Kap. 5, S. 58–76. – ISBN 3-540-53683-3 23, 24, 25

91. **Cunis und Günter 1991** Cunis, Roman ; Günter, Andreas: PLAKON – Übersicht über das System. In: (Cunis et al. 1991), Kap. 4, S. 37–57. – ISBN 3-540-53683-3 10, 12, 13, 20, 24, 32, 34

92. **Cunis et al. 1991** Cunis, Roman (Hrsg.) ; Günter, Andreas (Hrsg.) ; Strecker, Helmut (Hrsg.): *Das PLAKON-Buch, Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen.* Berlin, Heidelberg, New York : Springer Verlag, 1991 (Informatik-Fachberichte, Subreihe Künstliche Intelligenz 266). – 279 S. – ISBN 3-540-53683-3 13, 18, 349, 360, 373, 382

93. **Davis 1987** DAVIS, Ernest: Constraint Propagation with Interval Labels. In: *Artificial Intelligence* 32 (1987), Juli, Nr. 3, S. 281–331. – ISSN 0004-3702 2, 145, 147, 149, 150, 151, 152, 153, 154, 155, 156, 159, 172, 212, 225, 226, 313

94. **Debruyne 1998** DEBRUYNE, Romuald: Removing more Values than Max-Restricted Path Consistency for the same Cost / LIRMM, Université de Montpellier II. Montpellier, France, 1998 (Technical Report TR-98-041). – Forschungsbericht. – 21 S. – URL `http://debruyne.ifrance.com/debruyne/tr98041.pdf`. – Zugriffsdatum: 16. September 2004 110

95. **Debruyne 2000** DEBRUYNE, Romuald: A Property of Path Inverse Consistency Leading to an Optimal PIC Algorithm. In: HORN, Werner (Hrsg.): *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000), Berlin, 20.–25. August 2000.* Amsterdam, The Netherlands : IOS Press, 2000 (Frontiers in Artificial Intelligence and Applications 54), S. 88–92. – URL `http://debruyne.ifrance.com/debruyne/ecai2000.pdf`. – Zugriffsdatum: 19. September 2004. – ISBN 1-586-03013-2 108

96. **Debruyne und Bessière 1997a** DEBRUYNE, Romuald ; BESSIÈRE, Christian: From Restricted Path Consistency to Max-Restricted Path Consistency. In: SMOLKA, Gert (Hrsg.): *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97), Schloss Hagenberg, Linz, Austria, 29. Oktober – 1. November 1997.* Berlin, Heidelberg, New York : Springer Verlag, 1997 (LNCS 1330), S. 312–326. – URL `http://debruyne.ifrance.com/debruyne/cp97.pdf`. – Zugriffsdatum: 15. September 2004. – ISBN 3-540-63753-2 110

97. **Debruyne und Bessière 1997b** DEBRUYNE, Romuald ; BESSIÈRE, Christian: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In: (Pollack und Georgeff 1997), S. 412–417. – URL `http://debruyne.ifrance.com/debruyne/ijcai97.pdf`. – Zugriffsdatum: 15. September 2004. – ISBN 1-558-60480-4 110, 111

98. **Debruyne und Bessière 2001** DEBRUYNE, Romuald ; BESSIÈRE, Christian: Domain Filtering Consistencies. In: *Journal of Artificial Intelligence Research (JAIR)* 14 (2001), Mai, S. 205–230. – URL `http://www.emn.fr/x-info/rdebruyn/jair01.pdf`. – Zugriffsdatum: 21. September 2004 111

99. **Dechter 1990a** DECHTER, Rina: Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. In: *Artificial Intelligence* 41 (1990), Januar, Nr. 3, S. 273–312. – ISSN 0004-3702 89, 118, 125, 157

100. **Dechter 1990b** DECHTER, Rina: On the Expressiveness of Networks with Hidden Variables. In: DIETTERICH, Thomas (Hrsg.) ; SWARTOUT, William (Hrsg.): *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI'90), Boston, Massachusetts, USA, 29. Juli – 3. August 1990.* Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1990, S. 556–562. – URL

http://www.ics.uci.edu/~csp/r13.pdf. – Zugriffsdatum: 15. September 2004. – ISBN 0-262-51057-X 139

101. **Dechter 1992** Dechter, Rina: Constraint Networks. In: Shapiro, Stuart C. (Hrsg.): *Encyclopedia of Artificial Intelligence* Bd. 1. 2. Aufl. Chichester, London, New York : John Wiley & Sons, Februar 1992, S. 276–285. – URL http://www.ics.uci.edu/~csp/r17-survey.pdf. – Zugriffsdatum: 16. September 2004. – ISBN 0-4715-0307-X 83, 105, 118

102. **Dechter 1999** Dechter, Rina: Constraint Satisfaction. In: Wilson, Robert A. (Hrsg.) ; Keil, Frank C. (Hrsg.): *The MIT Encyclopedia of the Cognitive Sciences (MITECS).* Cambridge, Massachusetts, USA : Bradford Books/The MIT Press, 1. Juni 1999, S. 195–197. – URL http://www.ics.uci.edu/~csp/R68.pdf. – Zugriffsdatum: 16. September 2004. – ISBN 0-262-73124-X 57, 83, 87, 89, 118

103. **Dechter 2000** Dechter, Rina (Hrsg.): *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000), Singapore, 18.–21. September 2000.* Berlin, Heidelberg, New York : Springer Verlag, 2000. (LNCS 1894). – ISBN 3-540-41053-8 339, 362, 374

104. **Dechter und Dechter 1988** Dechter, Rina ; Dechter, Avi: Belief Maintenance in Dynamic Constraint Networks. In: Mitchell, Tom M. (Hrsg.) ; Smith, Reid G. (Hrsg.): *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI'88), St. Paul, Minnesota, USA, 21.–26. August 1988.* Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1988, S. 37–42. – URL http://www.ics.uci.edu/~csp/r5.pdf. – Zugriffsdatum: 20. Oktober 2004. – ISBN 0-262-51055-3 59

105. **Dechter und Frost 1998** Dechter, Rina ; Frost, Daniel: Backtracking Algorithms for Constraint Satisfaction Problems – a Tutorial Survey / University of California, Irvine (UCI), Information and Computer Science Department (ICS). Irvine, California, USA, 1998. – Forschungsbericht. – 57 S. – URL http://citeseer.ist.psu.edu/dechter98backtracking.html. – Zugriffsdatum: 20. September 2004. Überarb. Fassung: (Dechter und Frost 2002) 112, 116, 117, 123, 125, 134, 213, 311, 351

106. **Dechter und Frost 2002** Dechter, Rina ; Frost, Daniel: Backjump-based Backtracking for Constraint Satisfaction Problems. In: *Artificial Intelligence* 136 (2002), April, Nr. 2, S. 147–188. – Vorhergehende Version: (Dechter und Frost 1998). – ISSN 0004-3702 88, 112, 118, 120, 125, 134, 213, 311, 312, 351

107. **Dechter und Meiri 1994** Dechter, Rina ; Meiri, Itay: Experimental Evaluation of Preprocessing Algorithms for Constraint Satisfaction Problems. In: *Artificial Intelligence* 68 (1994), August, Nr. 2, S. 211–241. – Vorhergehende Version: (Sridharan 1989), S. 271–277. – ISSN 0004-3702 123, 129, 130, 131, 132

108. **Dechter et al. 1991** Dechter, Rina ; Meiri, Itay ; Pearl, Judea: Temporal Constraint Networks. In: *Artificial Intelligence* 49 (1991), Mai, Nr. 1–3, S. 61–95. – URL http://www.ics.uci.edu/~csp/r10.pdf. – Zugriffsdatum: 17. Oktober 2004. – Vorhergehende Version: Brachman, Ronald J. ; Levesque, Hector J. ; Reiter, Raymond: Proceedings of KR'89, Morgan Kaufmann Publishers, 1989, S. 83–93. – ISSN 0004-3702 58

109. **Dechter und Pearl 1987** Dechter, Rina ; Pearl, Judea: Network-Based Heuristics for Constraint-Satisfaction Problems. In: *Artificial Intelligence* 34 (1987), Dezember, Nr. 1, S. 1–38. – Eine leicht überarb. Fassung erschien in: Kanal, Laveen N. (Hrsg.) ; Kumar, Vipin (Hrsg.): Search in Artificial Intelligence, Springer Verlag, 1988, S. 370–425. – ISSN 0004-3702 105, 109, 110, 157

110. **Dechter und Pearl 1989** Dechter, Rina ; Pearl, Judea: Tree Clustering for Constraint Networks. In: *Artificial Intelligence* 38 (1989), April, Nr. 3, S. 353–366. – ISSN 0004-3702 89, 91, 105, 140, 157

111. **Dechter und Rossi 2003** Dechter, Rina ; Rossi, Francesca: Constraint Satisfaction. In: Nadel, Lynn (Hrsg.): *Encyclopedia of Cognitive Science (ECS)* Bd. 1. London, New York, Tokyo : Nature Publishing Group/Macmillan Publishers, Juli 2003, S. 793–780. – URL http://www.ics.uci.edu/~csp/r85.pdf. – Zugriffsdatum: 16. September 2004. – ISBN 0-333-79261-0 53, 83, 166

112. **Dent und Mercer 1994a** Dent, Michael J. ; Mercer, Robert E.: Minimal Forward Checking. In: *Proceedings of the 6th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'94), 6.–9. November 1994, New Orleans, Louisiana, USA*. Los Alamitos, California, USA, : IEEE Computer Society Press, 1994, S. 432–438. – URL http://www.csd.uwo.ca/tech-reports/374/tai94.ps.Z. – Zugriffsdatum: 16. September 2004. – Ausführliche Version: (Dent und Mercer 1994b). – ISBN 0-8186-6785-0 122, 352

113. **Dent und Mercer 1994b** Dent, Michael J. ; Mercer, Robert E.: Minimal Forward Checking / University of Western Ontario, Computer Science Department. London, Ontario, Canada, 3. Februar 1994 (Technical Report UWO-CSD-374). – Forschungsbericht. – 38 S. – URL http://www.csd.uwo.ca/tech-reports/374/minfc.techreport.ps.Z. – Zugriffsdatum: 16. September 2004. Revidierte und gekürzte Fassung: (Dent und Mercer 1994a) 122, 352

114. **Diaz und Codognet 2001** Diaz, Daniel ; Codognet, Philippe: Design and Implementation of the GNU Prolog System. In: *Journal of Functional and Logic Programming (JFLP)* 2001 (2001), Oktober, Nr. 6. – URL http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2001/S01-02/JFLP-A01-06.pdf. – Zugriffsdatum: 13. Januar 2005. – ISSN 1080-5230 67

115. **van Dongen 2002a** Dongen, Marc R. C. van: AC-3_d an Efficient Arc-Consistency Algorithm with a Low Space-Complexity / Cork Constraint Computation Centre.

Cork, Ireland, Juni 2002 (Technical Report TR-01-2002). – Forschungsbericht. – 22 S. – URL `http://csweb.ucc.ie/~dongen/papers/4C/02/4C-01-2002.pdf`. – Zugriffsdatum: 17. September 2004. Gekürzte Fassung: (van Dongen 2002b) 97, 106, 353

116. **van Dongen 2002b** DONGEN, Marc R. C. van: AC-3_d an Efficient Arc-Consistency Algorithm with a Low Space-Complexity. In: VAN HENTENRYCK, Pascal (Hrsg.): *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02), Ithaca, New York, USA, 8.–13. September 2002*. Berlin, Heidelberg, New York : Springer Verlag, 2002 (LNCS 2470), S. 755–760. – URL `http://csweb.ucc.ie/~dongen/papers/CP/02/cp02.pdf`. – Zugriffsdatum: 14. September 2004. – Ausführliche Version: (van Dongen 2002a). – ISBN 3-540-44120-4 97, 106, 353

117. **Donnelly und Stallman 2002** DONNELLY, Charles ; STALLMAN, Richard: Bison, The YACC-compatible Parser Generator / Free Software Foundation. Boston, Massachusetts, USA, 25. Februar 2002 (Version 1.35). – Manual. – iv + 101 S. – URL `ftp://prep.ai.mit.edu/pub/gnu/Manuals/bison/ps/bison.ps.gz`. – Zugriffsdatum: 8. November 2005 204

118. **Eckstein 2000** ECKSTEIN, Robert: *XML – kurz & gut*. 2., korrigierter Nachdruck. Köln : O'Reilly Verlag, Juli 2000. – 110 S. – ISBN 3-89721-219-6 200, 278

119. **Embacher und Oberhuemer 2003** EMBACHER, Franz (Hrsg.) ; OBERHUEMER, Petra (Hrsg.): *mathe online*. Universität Wien, Österreich : Future Media. Verein zur Förderung multimedialer Qualitätsprodukte, Stand: 7. November 2003. – URL `http://www.mathe-online.at`. – Zugriffsdatum: 8. April 2004 146, 164, 332, 333, 335

120. **Emde et al. 1996** EMDE, Werner ; BEILKEN, Christian ; BÖRDING, Josef ; ORTH, Wolfgang ; PETERSEN, Ulrike ; RAHMER, Jörg ; SPENKE, Michael ; VOSS, Angi ; WROBEL, Stefan: Configuration of Telecommunication Systems in KIKon. In: FALTINGS, Boi V. (Hrsg.) ; FREUDER, Eugene C. (Hrsg.): *Configuration – Papers from the AAAI Fall Symposium, Technical Report FS-96-03, Boston, Massachusetts, USA, 9.–11. November 1996*. Menlo Park, California, USA : AAAI Press, 1996, S. 105–110. – URL `http://citeseer.ist.psu.edu/263105.html`. – Zugriffsdatum: 14. September 2004. – ISBN 1-57735-018-9 15, 17, 266

121. **van Emden 2002** EMDEN, Maarten H. van: Interval Constraints / University of Victoria, Computer Science Department. British Columbia, Canada, Stand: 26. Dezember 2002. – Tutorial. – URL `http://csr.uvic.ca/~vanemden/research/intConstrInd.html`. – Zugriffsdatum: 19. April 2005 143

122. **Faltings 1994** FALTINGS, Boi V.: Arc-Consistency for Continuous Variables. In: *Artificial Intelligence* 65 (1994), Februar, Nr. 2, S. 363–376. – URL `http://liawww.epfl.ch/Publications/Archive/Faltings1994.pdf`. – Zugriffsdatum: 14. September 2004. – ISSN 0004-3702 51, 162

123. **Faltings et al. 1999** FALTINGS, Boi V. (Hrsg.) ; FREUDER, Eugene C. (Hrsg.) ; FRIEDRICH, Gerhard (Hrsg.) ; FELFERNIG, Alexander (Hrsg.): *Configuration – Papers from the AAAI Workshop at the 16th National Conference on Artificial Intelligence (AAAI'99), Orlando, Florida, USA, 18.–22. Juli 1999, AAAI Workshop Technical Report WS-99-05*. Menlo Park, California, USA : AAAI Press, 1999. – x + 140 S. – ISBN 1-57735-089-8 339, 354, 364, 372, 373, 379

124. **Faltings und Gelle 1997** FALTINGS, Boi V. ; GELLE, Esther: Local Consistency for Ternary Numeric Constraints. In: (Pollack und Georgeff 1997), S. 392–397. – URL `http://liawww.epfl.ch/Publications/Archive/Faltings1997d.pdf`. – Zugriffsdatum: 15. September 2004. – ISBN 1-558-60480-4 162

125. **Faltings und Macho-Gonzalez 2002** FALTINGS, Boi V. ; MACHO-GONZALEZ, Santiago: Open Constraint Satisfaction. In: VAN HENTENRYCK, Pascal (Hrsg.): *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02), Ithaca, New York, USA, 8.–13. September 2002*. Berlin, Heidelberg, New York : Springer Verlag, 2002 (LNCS 2470), S. 356–370. – URL `http://liawww.epfl.ch/Publications/Archive/Faltings2002a.pdf`. – Zugriffsdatum: 8. November 2004. – ISBN 3-540-44120-4 63

126. **Faltings und Macho-Gonzalez 2003** FALTINGS, Boi V. ; MACHO-GONZALEZ, Santiago: Open Constraint Optimization. In: ROSSI, Francesca (Hrsg.): *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03), Kinsale, Ireland, 29. September – 3. Oktober 2003*. Berlin, Heidelberg, New York : Springer Verlag, 2003 (LNCS 2833), S. 303–317. – URL `http://liawww.epfl.ch/Publications/Archive/Faltings2003b.pdf`. – Zugriffsdatum: 11. November 2004. – ISBN 3-540-20202-1 63

127. **Fayad und Schmidt 1997** FAYAD, Mohamed ; SCHMIDT, Douglas C.: Object-oriented application frameworks. In: *Communications of the ACM (CACM)* 40 (1997), Oktober, Nr. 10, S. 32–38. – ISSN 0001-0782 179

128. **Felfernig et al. 2002** FELFERNIG, Alexander ; FRIEDRICH, Gerhard ; JANNACH, Dietmar ; ZANKER, Markus: Semantic Configuration Web Services in the CAWICOMS Project. In: HORROCKS, Ian (Hrsg.) ; HENDLER, James A. (Hrsg.): *The Semantic Web – Proceedings of the 1st International Semantic Web Conference (ISWC'02), Sardinia, Italy, 9.–12. Juni 2002*. Berlin, Heidelberg, New York : Springer Verlag, 2002 (LNCS 2342), S. 192–205. – URL `http://www.cawicoms.org/publications/2002/2002-0132-AFGF.pdf`. – Zugriffsdatum: 15. September 2004. – ISBN 3-540-43760-6 270

129. **Fleischanderl 1999** FLEISCHANDERL, Gerhard: Overview of Configurators as Effective Tools for Corporate Knowledge Management. In: (Faltings et al. 1999), S. 110–113. – ISBN 1-57735-089-8 267

130. **Fleischanderl et al. 1998** FLEISCHANDERL, Gerhard ; FRIEDRICH, Gerhard E. ; HASELBÖCK, Alois ; SCHREINER, Herwig ; STUMPTNER, Markus: Configuring Large

Systems Using Generative Constraint Satisfaction. In: *IEEE Intelligent Systems* 13 (1998), Juli/August, Nr. 4, S. 59–68. – ISSN 1094-7167 18, 56, 268

131. **Fowler und Scott 2000** FOWLER, Martin ; SCOTT, Kendall: *UML konzentriert – Eine strukturierte Einführung in die Standard-Objektmodellierungssprache.* 2., aktual. Aufl. München : Addison-Wesley, 2000 (Professionelle Softwarewareentwicklung). – xvii + 179 S. – ISBN 3-8273-1617-0 201

132. **Frank et al. 2003a** FRANK, Stephan ; HOFSTEDT, Petra ; MAI, Pierre R.: A Flexible Meta-Solver Framework for Constraint Solver Collaboration. In: GÜNTER, Andreas (Hrsg.) ; KRUSE, Rudolf (Hrsg.) ; NEUMANN, Bernd (Hrsg.): *Proceedings of the 26th Annual German Conference on AI (KI'03): Advances in Artificial Intelligence, Hamburg, Germany, 15.–18. September 2003.* Berlin, Heidelberg, New York : Springer Verlag, 2003 (LNCS 2821), S. 520–534. – URL `http://uebb.cs.tu-berlin.de/~ph/ph.papers/ki2003.pdf`. – Zugriffsdatum: 24. Mai 2005. – ISBN 3-540-20059-2 78

133. **Frank et al. 2003b** FRANK, Stephan ; HOFSTEDT, Petra ; MAI, Pierre R.: Meta-S: A Strategy-Oriented Meta-Solver Framework. In: RUSSELL, Ingrid (Hrsg.) ; HALLER, Susan M. (Hrsg.): *Proceedings of the 16th International Florida Artificial Intelligence Research Society Conference (FLAIRS'03), Special Track on Constraint Solving and Programming, St. Augustine, Florida, USA, 12.–14. Mai 2003.* Menlo Park, California, USA : AAAI Press, 2003, S. 177–181. – URL `http://uebb.cs.tu-berlin.de/~ph/ph.papers/flairs2003.pdf`. – Zugriffsdatum: 24. Mai 2005. – ISBN 1-57735-177-0 78

134. **Freuder 1978** FREUDER, Eugene C.: Synthesizing Constraint Expressions. In: *Communications of the ACM (CACM)* 21 (1978), November, Nr. 11, S. 958–966. – ISSN 0001-0782 60, 85, 90, 91, 103, 105, 106, 107, 140

135. **Freuder 1982** FREUDER, Eugene C.: A Sufficient Condition for Backtrack-Free Search. In: *Journal of the ACM (JACM)* 29 (1982), Januar, Nr. 1, S. 24–32. – ISSN 0004-5411 104, 105, 129, 130, 157

136. **Freuder 1985** FREUDER, Eugene C.: A Sufficient Condition for Backtrack-Bounded Search. In: *Journal of the ACM (JACM)* 32 (1985), Oktober, Nr. 4, S. 755–761. – ISSN 0004-5411 108

137. **Freuder 1989** FREUDER, Eugene C.: Partial Constraint Satisfaction. In: (Sridharan 1989), S. 278–283. – Erweiterte Fassung: (Freuder und Wallace 1992), (Freuder und Mackworth 1994), S. 21–70 und (Jampel et al. 1996), S. 63–110. – ISBN 1-55860-094-9 60, 356

138. **Freuder 1995** FREUDER, Eugene C.: Using Metalevel Constraint Knowledge to Reduce Constraint Checking. In: MEYER, Manfred (Hrsg.): *Constraint Processing, Selected Papers.* Berlin, Heidelberg, New York : Springer Verlag, 1995 (LNCS 923), S. 171–184. – ISBN 3-540-59479-5 97

139. **Freuder 1996** FREUDER, Eugene C. (Hrsg.): *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP'96), Cambridge, Massachusetts, USA, 19.–22. August 1996*. Berlin, Heidelberg, New York : Springer Verlag, 1996. (LNCS 1118). – 572 S. – ISBN 3-540-61551-2 344, 357, 379

140. **Freuder 1997** FREUDER, Eugene C.: In Pursuit of the Holy Grail. In: *Constraints, An International Journal* 2 (1997), April, Nr. 1, S. 57–61. – Zugl.: ACM CSUR, 28 (1996), Nr. 4es, Artikel Nr. 63. – ISSN 1383-7133 v, 52

141. **Freuder und Elfe 1996** FREUDER, Eugene C. ; ELFE, Charles D.: Neighborhood Inverse Consistency Preprocessing. In: (Clancey et al. 1996), S. 202–208. – URL `http://www.cs.unh.edu/~cde/aaai96-nic-prep-ecf-cde.ps.gz`. – Zugriffsdatum: 15. September 2004. – ISBN 0-262-51091-X 108

142. **Freuder und Mackworth 1994** FREUDER, Eugene C. (Hrsg.) ; MACKWORTH, Alan K. (Hrsg.): *Constraint-Based Reasoning*. Cambridge, Massachusetts, USA : Bradford Books/The MIT Press, Februar 1994 (Special Issues of Artificial Intelligence). – 409 S. – ISBN 0-262-56075-5 355, 356, 363, 370, 371

143. **Freuder und Wallace 2000** FREUDER, Eugene C. ; WALLACE, Mark: Constraint Technology and the Commercial World (Interview). In: *IEEE Intelligent Systems* 15 (2000), Januar/Februar, Nr. 1, S. 20–23. – ISSN 1094-7167 69

144. **Freuder und Wallace 1992** FREUDER, Eugene C. ; WALLACE, Richard J.: Partial Constraint Satisfaction. In: *Artificial Intelligence* 58 (1992), Dezember, Nr. 1–3, S. 21–70. – Special Volume on Constraint Based Reasoning. – Zugl.: (Freuder und Mackworth 1994), S. 21–70 und (Jampel et al. 1996), S. 63–110. – Vorhergehende Version: (Freuder 1989). – ISSN 0004-3702 60, 61, 355

145. **Frost und Dechter 1994** FROST, Daniel ; DECHTER, Rina: In Search of the Best Constraint Satisfaction Search. In: HAYES-ROTH, Barbara (Hrsg.) ; KORF, Richard E. (Hrsg.): *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94), Seattle, Washington, USA, 31. Juli – 4. August 1994* Bd. 1. Menlo Park, California, USA : AAAI Press, August 1994, S. 301–306. – URL `http://www.ics.uci.edu/~csp/r35a-search-for-best-search.pdf`. – Zugriffsdatum: 15. September 2004. – ISBN 0-262-51078-2 118, 129, 132

146. **Frost und Dechter 1995** FROST, Daniel ; DECHTER, Rina: Look-Ahead Value Ordering for Constraint Satisfaction Problems. In: PERRAULT, C. Raymond (Hrsg.) ; MELLISH, Chris S. (Hrsg.): *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95), Montréal, Québec, Canada, 20.–25. August 1995* Bd. 1. San Mateo, California, USA : Morgan Kaufmann Publishers, Dezember 1995, S. 572–578. – URL `http://www.ics.uci.edu/~csp/r39-look-ahead-value-ordering.pdf`. – Zugriffsdatum: 10. Oktober 2004. – ISBN 1-558-60363-8 132, 133, 134, 135

147. **Frost und Dechter 1996a** Frost, Daniel ; Dechter, Rina: Looking at Full Looking Ahead / University of California, Irvine (UCI), Information and Computer Science Department (ICS). Irvine, California, USA, 1996. – Forschungsbericht. – 14 S. – URL `http://www.ics.uci.edu/~csp/dan-CP96-full.pdf`. – Zugriffsdatum: 16. September 2004. Gekürzte Fassung: (Frost und Dechter 1996b) 123, 125, 357

148. **Frost und Dechter 1996b** Frost, Daniel ; Dechter, Rina: Looking at Full Looking Ahead. In: (Freuder 1996), S. 539–540 (Poster). – URL `http://citeseer.ist.psu.edu/frost96looking.html`. – Zugriffsdatum: 16. September 2004. – Ausführliche Version: (Frost und Dechter 1996a). – ISBN 3-540-61551-2 125, 357

149. **Frühwirth 1995** Frühwirth, Thom: Constraint Handling Rules. In: (Podelski 1995), S. 90–107. – URL `http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/Papers/chatillon.ps.gz`. – Zugriffsdatum: 10. März 2005. – ISBN 3-540-59155-9 75

150. **Frühwirth 1998** Frühwirth, Thom: Theory and Practice of Constraint Handling Rules. In: *The Journal of Logic Programming* 37 (1998), Oktober, Nr. 1-3, S. 95–138. – URL `http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/drafts/jlp-chr1.ps.Z`. – Zugriffsdatum: 10. März 2005. – Special Issue on Constraint Logic Programming. – ISSN 0743-1066 75, 76

151. **Frühwirth und Abdennadher 1997** Frühwirth, Thom ; Abdennadher, Slim: *Constraint-Programmierung, Grundlagen und Anwendungen*. Berlin, Heidelberg, New York : Springer Verlag, 1997. – ix + 165 S. – ISBN 3-540-60670-X 44, 63, 66, 67, 75

152. **Frühwirth et al. 1992** Frühwirth, Thom ; Herold, Alexander ; Küchenhoff, Volker ; Le Provost, Thierry ; Lim, Pierre ; Monfroy, Eric ; Wallace, Mark: Contraint Logic Programming – An Informal Introduction. In: Comyn, Gérard (Hrsg.) ; Fuchs, Norbert E. (Hrsg.) ; Ratcliffe, Michael J. (Hrsg.): *Logic Programming in Action, Proceedings of the 2nd International Logic Programming Summer School (LPSS'92), Zurich, Switzerland, 7.–11. September 1992*. Berlin, Heidelberg, New York : Springer Verlag, 1992 (LNCS 636), S. 3–35. – URL `http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/Papers/ecrc91-95/ECRC-93-05.ps.gz`. – Zugriffsdatum: 22. Dezember 2004. – Zugl.: Technical Report ECRC-93-5, ECRC Munich, Germany, February 1993. – ISBN 3-540-55930-2 66

153. **Gamma et al. 1996** Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl. München : Addison-Wesley, 1996. – xx + 479 S. – ISBN 3-89319-950-0 178, 200, 201, 203, 206, 208, 209, 330

154. **Gaschnig 1974** Gaschnig, John G.: A Constraint Satisfaction Method for Inference Making. In: *Proceedings of the 12th Annual Allerton Conference on Circuit*

and System Theorie. University of Illinois, Urbana-Champaign, USA, Oktober 1974, S. 866–874. – ISSN 0569-0552 123

155. **Gaschnig 1979** GASCHNIG, John G.: *Performance Measurement and Analysis of Certain Search Algorithms*. Pittsburgh, Pennsylvania, USA, Carnegie Mellon University, Computer Science Department, PhD Thesis, 1979. – 330 S. – (Technical Report CMU-CS-79-124) 118, 123

156. **Gelle 1998** GELLE, Esther: *On the Generation of Locally Consistent Solution Spaces in Mixed Dynamic CSPs*. Lausanne (Switzerland), Swiss Federal Institute of Technology (EPFL), PhD. Thesis No. 1826, 1998. – xx + 180 S. – URL `http://liawww.epfl.ch/Publications/Archive/Gelle1998.pdf`. – Zugriffsdatum: 18. November 2004 65, 77, 190

157. **Gelle und Faltings 2003** GELLE, Esther ; FALTINGS, Boi V.: Solving Mixed and Conditional Constraint Satisfaction Problems. In: *Constraints, An International Journal* 8 (2003), April, Nr. 2, S. 107–141. – URL `http://liawww.epfl.ch/Publications/Archive/Gelle2003.pdf`. – Zugriffsdatum: 10. November 2004. – ISSN 1383-7133 64, 65, 77, 190

158. **Gennari 1998** GENNARI, Rosella: Temporal Reasoning and Constraint Programming – A Survey. In: *CWI Quarterly* 11 (1998), Juni/September, Nr. 2–3, S. 163–214. – URL `ftp://ftp.cwi.nl/pub/CWIQuarterly/1998/11.2-3/gennari.pdf`. – Zugriffsdatum: 17. Oktober 2004. – ISSN 0168-826X 58

159. **Gent und Prosser 2000** GENT, Ian P. ; PROSSER, Patrick: Inside MAC and FC / Algorithms, Problems and Empirical Studies Research Group (APES). Alberta, Leeds, St. Andrews, Strathclyde, York, Mai 2000 (Research Report APES-20-2000). – Forschungsbericht. – 12 S. – URL `http://www.dcs.st-and.ac.uk/~apes/reports/apes-20-2000.ps.gz`. – Zugriffsdatum: 20. September 2004 123, 125

160. **Ginsberg 1993** GINSBERG, Matthew L.: Dynamic Backtracking. In: *Journal of Artificial Intelligence Research (JAIR)* 1 (1993), August, S. 25–46. – URL `http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume1/ginsberg93a.pdf`. – Zugriffsdatum: 25. Oktober 2004 151

161. **Godehardt und Seifert 2001** GODEHARDT, Eicke ; SEIFERT, Dirk: *Kooperation und Koordination von Constraint Solvern – Implementierung eines Prototyps*, Technische Universität Berlin, Institut für Kommunikations- und Softwaretechnik, Fachgebiet Übersetzerbau und Programmiersprachen, Diplomarbeit, 22. Januar 2001. – URL `http://swt.cs.tu-berlin.de/~seifert/pub/Diplomarbeit.ps`. – Zugriffsdatum: 19. Juli 2005 78

162. **Grant und Smith 1995** GRANT, Stuart A. ; SMITH, Barbara M.: The Phase Transition Behaviour of Maintaining Arc Consistency / University of Leeds, School of Computer Studies. Leeds, UK, August 1995 (Research Report Series, Report 95–25). – Forschungsbericht. – 46 S. – URL `http://scom.hud.ac.uk/staff/scombms/Papers/`

ResearchReports/95_25.ps.gz. – Zugriffsdatum: 16. September 2004. Revidierte und gekürzte Fassung: (Grant und Smith 1996) 57, 123, 124, 359

163. **Grant und Smith 1996** GRANT, Stuart A. ; SMITH, Barbara M.: The Phase Transition Behaviour of Maintaining Arc Consistency. In: WAHLSTER, Wolfgang (Hrsg.): *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI'96), Budapest, Hungary, 11.–16. August 1996.* Chichester, London, New York : John Wiley & Sons, 1996, S. 175–179. – Ausführliche Version: (Grant und Smith 1995). – ISBN 0-471-96809-9 57, 125, 126, 359

164. **Granvilliers 2001** GRANVILLIERS, Laurent: On the Combination of Interval Constraint Solvers. In: *Reliable Computing* 7 (2001), Dezember, Nr. 6, S. 467–483. – ISSN 1385-3139 77

165. **Granvilliers 2004** GRANVILLIERS, Laurent: RealPaver User's Manual – Solving Nonlinear Constraints by Interval Computations / Université de Nantes, Laboratoire d'Informatique de Nantes Atlantique. France, August 2004 (Edition 0.4, for RealPaver Version 0.4). – Handbuch. – iv + 30 S. – URL http://sourceforge.net/projects/realpaver. – Zugriffsdatum: 11. Februar 2005 70

166. **Granvilliers et al. 1999** GRANVILLIERS, Laurent ; GOUALARD, Frédéric ; BENHAMOU, Frédéric: Box Consistency through Weak Box Consistency. In: MENG, Weiyi (Hrsg.): *Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'99), 8.–10. November, Chicago, Illinois, USA.* Los Alamitos, California, USA, : IEEE Computer Society Press, 1999, S. 373–380. – URL http://goualard.free.fr/publications/files/Granvilliers-et-al-ICTAI99.pdf. – Zugriffsdatum: 27. Mai 2005. – ISBN ISBN 0-7695-0456-6 167

167. **Granvilliers et al. 2001a** GRANVILLIERS, Laurent ; MONFROY, Eric ; BENHAMOU, Frédéric: Cooperative Solvers in Constraint Programming: A Short Introduction. In: *ALP Newsletter (Association for Logic Programming)* 14 (2001), Mai, Nr. 2. – URL http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/may01/nav/cooperation/cooperation.ps. – Zugriffsdatum: 12. Juni 2005. – Zugl.: (Granvilliers et al. 2001b) 77, 359

168. **Granvilliers et al. 2001b** GRANVILLIERS, Laurent ; MONFROY, Eric ; BENHAMOU, Frédéric: Cooperative Solvers in Constraint Programming: A Short Introduction. In: MONFROY, Eric (Hrsg.) ; GRANVILLIERS, Laurent (Hrsg.): *Proceedings of the Workshop on Cooperative Solvers in Constraint Programming (CoSolv'01) at the 7th International Conference on Principles and Practice of Constraint Programming (CP'01).* Paphos, Zypern, 1. Dezember 2001, S. 1–6. – URL http://www.sciences.univ-nantes.fr/info/recherche/Theme_Contraintes/cosolv/Papers/Granvilliers_Monfroy_Benhamou.pdf. – Zugriffsdatum: 1. Juni 2005. – Zugl.: (Granvilliers et al. 2001a) 77, 359

169. **Granvilliers et al. 2001c** GRANVILLIERS, Laurent ; MONFROY, Eric ; BENHAMOU, Frédéric: Symbolic-Interval Cooperation in Constraint Programming

(Survey Paper). In: *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation (ISSAC'01), London, Ontario, Canada, 22.–25. Juli 2001.* New York, NY, USA : ACM Press, 2001, S. 150–166. – URL `http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/GraMonBen_ISSAC2001.pdf`. – Zugriffsdatum: 9. Juni 2005. – ISBN 1-58113-417-7 77

170. **Gülcü 2002** GÜLCÜ, Ceki: Short introduction to log4j / The Apache Software Foundation. Forest Hill, Maryland, USA, März 2002. – Manual. – URL `http://logging.apache.org/log4j/docs/manual.html`. – Zugriffsdatum: 7. November 2005. Überarb. Fassung: (Schnelle 2004) 200, 380

171. **Gülden 1993** GÜLDEN, Oliver: *Entwurf und Implementierung eines prototypischen Constraintsystems für das Konfigurierungswerkzeug KONWERK*, Universität Hamburg, Fachbereich Informatik, Studienarbeit, Dezember 1993. – iii + 76 S. – (PROKON-Memo Nr. 46) 2, 34, 43, 57, 61

172. **Günter 1991a** GÜNTER, Andreas: Begriffshierarchie-orientierte Kontrolle. In: (Cunis et al. 1991), Kap. 7, S. 92–110. – ISBN 3-540-53683-3 12, 13, 29, 30, 31, 32, 275

173. **Günter 1991b** GÜNTER, Andreas: Expertensysteme für Konstruktionsaufgaben. In: (Cunis et al. 1991), S. 1–3. – ISBN 3-540-53683-3 9, 10

174. **Günter 1992** GÜNTER, Andreas: *Flexible Kontrolle in Expertensystemen zur Planung und Konfigurierung in technischen Domänen.* Sankt Augustin : Infix Verlag, 1992 (DISKI 3). – iv + 239 S. – Zugl.: Hamburg, Universität, Dissertation, 1991. – ISBN 3-929037-03-3 8, 11, 12, 13, 23, 29, 31, 33, 39, 41, 42, 44, 60, 261, 272

175. **Günter 1995a** GÜNTER, Andreas: KONWERK – ein modulares Konfigurierungswerkzeug. In: RICHTER, Michael M. (Hrsg.) ; MAURER, Frank (Hrsg.): *Expertensysteme 95, Beiträge zur 3. Deutschen Expertensystemtagung (XPS'95), Kaiserslautern, 1.–3. März 1995.* Sankt Augustin : Infix Verlag, 1995, S. 1–18. – URL `http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/konwerk.pdf`. – Zugriffsdatum: 15. September 2004. – ISBN 3-932-79295-5 13, 18, 20

176. **Günter 1995b** GÜNTER, Andreas (Hrsg.): *Wissensbasiertes Konfigurieren – Ergebnisse aus dem Projekt PROKON.* Sankt Augustin : Infix Verlag, 1995. – viii + 367 S. – ISBN 3-929037-96-3 20, 23, 24, 52, 60

177. **Günter et al. 2001** GÜNTER, Andreas ; HOLLMANN, Oliver ; RANZE, K. Christoph ; WAGNER, Thomas: Wissensbasierte Konfiguration von komplexen variantenreichen Produkten in internetbasierten Vertriebsszenarien. In: *Künstliche Intelligenz* 15 (2001), März, Nr. 1, S. 33–36. – URL `http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/konfig_ec.pdf`. – Zugriffsdatum: 15. September 2004. – ISSN 0933-1875 21

178. **Günter et al. 1999** GÜNTER, Andreas ; KREUZ, Ingo ; KÜHN, Christan: Kommerzielle Software-Werkzeuge für die Konfigurierung von technischen Systemen. In: *Künstliche Intelligenz* 13 (1999), September, Nr. 3, S. 61–65. – URL `http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/konfig-tools.pdf`. – Zugriffsdatum: 15. September 2004. – ISSN 0933-1875 15, 17, 263, 264, 265, 266

179. **Günter und Kühn 1999** GÜNTER, Andreas ; KÜHN, Christian: Knowledge-Based Configuration – Survey and Future Directions. In: PUPPE, Frank (Hrsg.): *Knowledge-Based Systems – Survey and Future Directions, Proceedings of the 5th Biannual German Conference on Knowledge-Based Systems (XPS'99), Würzburg, 3.–5. März 1999*. Berlin, Heidelberg, New York : Springer Verlag, 1999 (LNCS 1570), S. 47–66. – URL `http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/xps-99.pdf`. – Zugriffsdatum: 8. Oktober 2004. – ISBN 3-540-65658-8 1, 11, 12, 14, 15, 17

180. **Güsgen 1989** GÜSGEN, Hans W.: *CONSAT – A System for Constraint Satisfaction*. San Mateo, California, USA : Morgan Kaufmann Publishers, Dezember 1989 (Research Notes in Artificial Intelligence). – 178 S. – Zugl.: Kaiserslautern, Universität, Dissertation, 1988. – ISBN 1-558-60093-0 34, 61, 85, 86

181. **Güsgen 2000** GÜSGEN, Hans W.: Constraints. In: GÖRZ, Günther (Hrsg.) ; ROLLINGER, Claus-Rainer (Hrsg.) ; JOSEF, Schneeberger (Hrsg.): *Handbuch der Künstlichen Intelligenz*. 3., vollst. überarb. Aufl. München, Wien : Oldenbourg Verlag, 2000, Kap. 8, S. 267–287. – ISBN 3-486-25049-3 51, 52, 60, 62, 83, 85, 86, 87, 88, 91, 92, 103, 104, 111, 113, 118, 123, 126, 128, 129, 134, 138, 143

182. **Haag 1998** HAAG, Albert: Sales Configuration in Business Processes. In: *IEEE Intelligent Systems* 13 (1998), Juli/August, Nr. 4, S. 78–85. – ISSN 1094-7167 266

183. **Han und Lee 1988** HAN, Ching-Chih ; LEE, Chia-Hoang: Comments on Mohr and Henderson's Path Consistency Algorithm. In: *Artificial Intelligence* 36 (1988), August, Nr. 1, S. 125–130. – ISSN 0004-3702 102, 106

184. **Haralick und Elliot 1980** HARALICK, Robert M. ; ELLIOT, Gordon L.: Increasing Tree Search Efficiency for Constraint Satisfaction Problems. In: *Artificial Intelligence* 14 (1980), Oktober, Nr. 3, S. 263–313. – ISSN 0004-3702 120, 121, 122, 123, 124, 125, 126, 127, 129

185. **Haralick und Shapiro 1979** HARALICK, Robert M. ; SHAPIRO, Linda G.: The Consistent Labeling Problem: Part I. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* 1 (1979), April, Nr. 2, S. 173–184. – ISSN 0162-8828 84, 91

186. **Haroud und Faltings 1994** HAROUD, Djamila ; FALTINGS, Boi V.: Global Consistency for Continuous Constraints. In: BORNING, Alan (Hrsg.): *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94), Rosario, Orcas Island, Washington, USA, 2.–4. Mai 1994*.

Berlin, Heidelberg, New York : Springer Verlag, 1994 (LNCS 874), S. 40–50. – URL http://liawww.epfl.ch/Publications/Archive/Haroud1994.pdf. – Zugriffsdatum: 22. Januar 2006. – Zugl.: Cohn, Anthony G. (Hrsg.): Proceedings of ECAI'94, John Wiley & Sons, 1994, S. 115–119. – ISBN 3-540-58601-6 143, 167, 168, 169, 173

187. **Heinrich und Jüngst 1993** HEINRICH, Michael ; JÜNGST, Ernst-Werner: Konfigurieren technischer Einrichtungen ausgehend von den Komponenten des technischen Prozesses: Prinzip und erste Erfahrungen. In: (Puppe und Günter 1993), S. 98–111. – ISBN 3-540-56464-0 15, 265

188. **Herold 1999** HEROLD, Helmut: *Linux-Unix-Profitools: awk, sed, lex, yacc und make*. 3., überarb. Aufl. Bonn : Addison-Wesley, 1999 (Linux/Unix und seine Werkzeuge). – vi + 882 S. – ISBN 3-8273-1448-8 205

189. **Hickey 2000** HICKEY, Timothy J.: CLIP: A CLP(Intervals) Dialect for Metalevel Constraint Solving. In: PONTELLI, Enrico (Hrsg.) ; SANTOS COSTA, Vítor (Hrsg.): *Proceedings of the 2nd International Workshop on Practical Aspects of Declarative Languages (PADL 2000), Boston, Massachusetts, USA 17.–18. Januar 2000*. Berlin, Heidelberg, New York : Springer Verlag, 2000 (LNCS 1753), S. 200–214. – ISBN 3-540-66992-2 68

190. **Hickey et al. 2000** HICKEY, Timothy J. ; QIU, Zhe ; EMDEN, Maarten H. van: Interval Constraint Plotting for Interactive Visual Exploration of Implicitly Defined Relations. In: *Reliable Computing* 6 (2000), Februar, Nr. 1, S. 81–92. – URL http://csr.uvic.ca/~vanemden/Publications/graphics.pdf. – Zugriffsdatum: 14. April 2005. – ISSN 1385-3139 72

191. **Hoche et al. 2003** HOCHE, Matthias ; MÜLLER, Henry ; SCHLENKER, Hans ; WOLF, Armin: firstcs – A Pure Java Constraint Programming Engine. In: HANUS, Michael (Hrsg.) ; HOFSTEDT, Petra (Hrsg.) ; WOLF, Armin (Hrsg.): *Proccedings of the 2nd International Workshop on Multiparadigm Constraint Programming Languages (MultiCPL'03) at the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*. Kinsale, County Cork, Ireland, 29. September 2003. – URL http://uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf. – Zugriffsdatum: 13. März 2005 75

192. **Hofstedt 2000** HOFSTEDT, Petra: Cooperating Constraint Solvers. In: (Dechter 2000), S. 520–524. – URL http://uebb.cs.tu-berlin.de/~ph/ph.papers/cp2000.pdf. – Zugriffsdatum: 24. Mai 2005. – ISBN 3-540-41053-8 78, 249, 250

193. **Hofstedt 2001** HOFSTEDT, Petra: *Cooperation and Coordination of Constraint Solvers*. Aachen : Shaker Verlag, September 2001 (Berichte aus der Informatik 124). – xiv + 222 S. – URL http://uebb.cs.tu-berlin.de/~ph/ph.papers/Hofstedt.PhDThesis.2001.pdf. – Zugriffsdatum: 24. Mai 2005. – Zugl.: Dresden, Technische Universität, Fakultät Informatik, Dissertation, 2001. – ISBN 3-8265-9351-0 78

194. **Hofstedt et al. 2001** HOFSTEDT, Petra ; SEIFERT, Dirk ; GODEHARDT, Eicke: A Framework for Cooperating Constraint Solvers – A Prototypic Implementation. In: MONFROY, Eric (Hrsg.) ; GRANVILLIERS, Laurent (Hrsg.): *Proceedings of the Workshop on Cooperative Solvers in Constraint Programming (CoSolv'01) at the 7th International Conference on Principles and Practice of Constraint Programming (CP'01).* Paphos, Zypern, 1. Dezember 2001, S. 7–21. – URL `http://uebb.cs.tu-berlin.de/~ph/ph.papers/cosolv2001.pdf`. – Zugriffsdatum: 24. Mai 2005 78

195. **Hollman und Langemyr 1993** HOLLMAN, Joachim ; LANGEMYR, Lars: Algorithms for Non-Linear Algebraic Constraints. In: BENHAMOU, Frédéric (Hrsg.) ; COLMERAUER, Alain (Hrsg.): *Constraint Logic Programming – Selected Research.* Cambridge, Massachusetts, USA : The MIT Press, 1993, Kap. 7, S. 113–131. – ISBN 0-262-02353-9 66, 83

196. **Hollmann et al. 2000** HOLLMANN, Oliver ; WAGNER, Thomas ; GÜNTER, Andreas: EngCon – A Flexible Domain-Independent Configuration Engine. In: *Proceedings of the Workshop on Configuration at the 14th European Conference on Artificial Intelligence (ECAI 2000).* Humboldt-Universität zu Berlin, 21.–22. August 2000, S. 94–96. – URL `http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/ecai2000.pdf`. – Zugriffsdatum: 15. September 2004 1, 13, 18, 19, 21

197. **Hudson 1999** HUDSON, Scott E.: CUP User's Manual / Georgia Institute of Technology, Graphics Visualization and Usability Center. Atlanta, Georgia, USA, Juli 1999 (Version 0.10j). – Online Handbuch. – URL `http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`. – Zugriffsdatum: 14. September 2005 200, 204, 282

198. **Hyvönen 1989** HYVÖNEN, Eero: Constraint Reasoning Based on Interval Arithmetic. In: (Sridharan 1989), S. 1193–1198. – ISBN 1-55860-094-9 143, 156, 158

199. **Hyvönen 1991** HYVÖNEN, Eero: Global Consistency in Interval Constraint Satisfaction. In: MAYOH, Brian H. (Hrsg.): *Proceedings of the 3rd Scandinavian Conference Conference on Artificial Intelligence (SCAI'91), Roskilde, Denmark, 21.–24. Mai 1991* Bd. 12. Amsterdam, The Netherlands : IOS Press, 1991, S. 241–251. – ISBN 90-5199-056-1 143, 157, 159

200. **Hyvönen 1992** HYVÖNEN, Eero: Constraint Reasoning Based on Interval Arithmetic: The Tolerance Propagation Approach. In: *Artificial Intelligence* 58 (1992), Dezember, Nr. 1–3, S. 71–112. – Special Volume on Constraint Based Reasoning. – Zugl.: (Freuder und Mackworth 1994), S. 71–112. – ISSN 0004-3702 2, 38, 58, 143, 145, 147, 148, 154, 155, 156, 157, 158, 159, 160, 173, 212, 313

201. **ILOG 2001** ILOG Configurator / ILOG S.A. Gentilly Cedex, France, Januar 2001. – Technical White Paper. – 28 S. – URL `http://www.ilog.com/products/configurator/whitepaper/index.cfm?filename=wp_configurator.pdf`. – Zugriffsdatum: 3. August 2004 18, 268

202. **ILOG 2002** ILOG JSolver – Constraint Programming for the Java Platform / ILOG S.A. Gentilly Cedex, France, Januar 2002. – Datasheet. – 2 S. – URL `http://www.ilog.com/products/jsolver/DS-JSolver.pdf`. – Zugriffsdatum: 7. Februar 2005 69

203. **ILOG 2003** ILOG JConfigurator / ILOG S.A. Gentilly Cedex, France, Juni 2003. – Technical White Paper. – 24 S. – URL `http://www.ilog.com/products/jconfigurator/whitepaper/index.cfm?filename=JConfiguratorWP.pdf`. – Zugriffsdatum: 3. August 2004 18, 268

204. **ILOG 2004a** ILOG CPLEX – Premier choice for resource optimization solutions / ILOG S.A. Gentilly Cedex, France, Juni 2004. – Datasheet. – 4 S. – URL `http://www.ilog.com/products/cplex/DS-CPLEX2004.pdf`. – Zugriffsdatum: 20. Februar 2005 69

205. **ILOG 2004b** ILOG JSolver – Optimal scheduling, dispatching and configuration / ILOG S.A. Gentilly Cedex, France, Oktober 2004. – Datasheet. – 2 S. – URL `http://www.ilog.com/products/solver/DS-Solver2004.pdf`. – Zugriffsdatum: 7. Februar 2005 69

206. **Jaffar 1999** Jaffar, Joxan (Hrsg.): *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99), Alexandria, Virginia, USA, 11.–14. Oktober 1999.* Berlin, Heidelberg, New York : Springer Verlag, 1999. (LNCS 1713). – 491 S. – ISBN 3-540-66626-5 343, 344, 349, 380

207. **Jaffar und Maher 1994** Jaffar, Joxan ; Maher, Michael J.: Constraint Logic Programming: A Survey. In: *The Journal of Logic Programming* 19/20 (1994), S. 503–581. – URL `ftp://ftp.cse.ohio-state.edu/pub/clp/papers/clp-survey.ps`. – Zugriffsdatum: 26. Januar 2006. – Special Issue: Ten Years of Logig Programming. – ISSN 0743-1066 53

208. **Jampel et al. 1996** Jampel, Michael (Hrsg.) ; Freuder, Eugene C. (Hrsg.) ; Maher, Michael J. (Hrsg.): *Over-Constrained Systems – Proceedings of the Workshop on Over-Constrained Systems (OCS'95) at the 1st International Conference on Principles and Practice of Constraint Programming (CP'95), Cassis, Marseilles, France, 18. September 1995.* Berlin, Heidelberg, New York : Springer Verlag, 1996. (LNCS 1106). – ISBN 3-540-61479-6 345, 355, 356

209. **John und Geske 1999a** John, Ulrich ; Geske, Ulrich: Constraint-logische Modellierung und Bearbeitung technischer Konfigurationsprobleme – Das System ConBaCon. In: *Beiträge zum 13. Workshop „Planen und Konfigurieren“ (PuK'99) im Rahmen der 5. Deutschen Expertensystemtagung (XPS'99).* Universität Würzburg, 3.–5. März 1999, S. 10. – URL `http://www.informatik.uni-freiburg.de/~koehler/puk99-papers/geske.ps.gz`. – Zugriffsdatum: 14. September 2004 14, 269

210. **John und Geske 1999b** John, Ulrich ; Geske, Ulrich: Reconfiguration of Technical Products Using ConBaCon. In: (Faltings et al. 1999), S. 48–53. – ISBN 1-57735-089-8 269

211. **John und Geske 2001** John, Ulrich ; Geske, Ulrich: Constraint-Based Configuration of Large Systems. In: Bartenstein, Oskar (Hrsg.) ; Geske, Ulrich (Hrsg.) ; Hannebauer, Markus (Hrsg.) ; Yoshie, Osama (Hrsg.): *Web Knowledge Management and Decision Support, 14th International Conference on Applications of Prolog (INAP'01), Tokyo, Japan, 20.–22. Oktober 2001, Revised Papers.* Berlin, Heidelberg, New York : Springer Verlag, 2001 (LNCS 2543), S. 217–234. – ISBN 3-540-00680-X 269

212. **John und Geske 2002** John, Ulrich ; Geske, Ulrich: Konfiguration komplexer Produkte mit Constraint-basierter Modellierung. In: *Informatik – Forschung und Entwicklung* 17 (2002), Dezember, Nr. 4, S. 167–176. – ISSN 0178-3564 15, 18, 269

213. **Johnson 1997a** Johnson, Ralph E.: Components, Frameworks, Patterns. In: Harandi, Medhi (Hrsg.): *Proceedings of the 1997 Symposium on Software Reusability (SSR'97), Boston, Massachusetts, USA, 17.–20. Mai 1997.* New York, NY, USA : ACM Press, 1997, S. 10–17. – URL `ftp://st.cs.uiuc.edu/pub/papers/frameworks/framework97.ps`. – Zugriffsdatum: 5. Januar 2006. – Zugl.: ACM SIGSOFT Software Engineering Notes, 22 (1997), Nr. 3, S. 10–17. – ISBN 0-89791-945-9 177

214. **Johnson 1997b** Johnson, Ralph E.: Frameworks = (Components + Patterns). In: *Communications of the ACM (CACM)* 40 (1997), Oktober, Nr. 10, S. 39–42. – ISSN 0001-0782 177

215. **Johnson 1975** Johnson, Stephen C.: Yacc: Yet Another Compiler-Compiler / AT&T Bell Laboratories. Murray Hill, New Jersey, USA, Juli 1975 (Computer Science Technical Report No. 32). – Forschungsbericht. – 33 S. – URL `http://dinosaur.compilertools.net/yacc/yacc.ps`. – Zugriffsdatum: 8. November 2005 204

216. **Johnson-Schaaf 1999** Johnson-Schaaf, Jörg W.: Kundenindividuelles Konfigurieren. In: *Der GMD-Spiegel* (1999), Nr. 1–2. – URL `http://www.gmd.de/de/GMD-Spiegel/GMD-Spiegel-1_2_99-html/pdf-version/Kikon.pdf`. – Zugriffsdatum: 16. September 2004. – ISSN 0724-4339 266

217. **Jüngst und Heinrich 1998** Jüngst, Ernst-Werner ; Heinrich, Michael: Using Resource Balancing to Configure Modular Systems. In: *IEEE Intelligent Systems* 13 (1998), Juli/August, Nr. 4, S. 50–58. – ISSN 1094-7167 15

218. **Junker und Mailharro 2003** Junker, Ulrich ; Mailharro, Daniel: The Logic of ILOG (J)Configurator: Combining Constraint Programming with a Description Logic. In: Mailharro, Daniel (Hrsg.): *Proceedings of the Workshop on Configuration at the 18th International Joint Conference on Artificial Intelligence (IJCAI'03).* Acapulco, Mexico, 11.August 2003, S. 13–20. – URL `http://www2.ilog.com/ijcai-03/Papers/IJCAI03-03.pdf`. – Zugriffsdatum: 16. September 2004 268

219. **Jussien und Lhomme 1998** Jussien, Narendra ; Lhomme, Olivier: Dynamic Domain Splitting for Numeric CSPs. In: Prade, Henri (Hrsg.): *Proceedings of the*

13th European Conference on Artificial Intelligence (ECAI'98), Brighton, UK, 23.–28. August 1998. Chichester, London, New York : John Wiley & Sons, 1998, S. 224–228. – URL `http://www.emn.fr/x-info/jussien/publications/jussien-ECAI98.pdf`. – Zugriffsdatum: 18. September 2004. – ISBN 0-471-98431-0 150

220. **Jussien et al. 1997** JUSSIEN, Narendra ; LHOMME, Olivier ; BOIZUMAULT, Patrice: Dynamic Backtracking with Constraint Propagation. In: *Journée du Pôle Contraintes et Programmation Logique*. Orléans, France : Groupe de Recherche Programmation du Centre National de la Recherche Scientifique (GDR Programmation du CNRS), 12.–14. November 1997. – URL `http://www.emn.fr/jussien/publications/jussien-GDR97.pdf`. – Zugriffsdatum: 20. September 2004 150

221. **Ka Boon et al. 2000** KA BOON, Ng ; CHUI, Wo Choi ; HENZ, Martin ; MÜLLER, Tobias: GIFT: A Generic Interface for reusing Filtering Algorithms. In: BELDICEANU, Nicolas (Hrsg.) ; HARVEY, Warwick (Hrsg.) ; HENZ, Martin (Hrsg.) ; LABURTHE, François (Hrsg.) ; MONFROY, Eric (Hrsg.) ; MÜLLER, Tobias (Hrsg.) ; PERRON, Laurent (Hrsg.) ; SCHULTE, Christian (Hrsg.): *Proceedings of the 1st Workshop on Techniques for Implementing Constraint Programming Systems (TRICS 2000), a post-conference Workshop of CP 2000 (veröffentlicht als technischer Bericht TRA9/00)*. National University of Singapore, School of Computing, September 2000, S. 86–100. – URL `http://www.comp.nus.edu.sg/~henz/publications/ps/gift.ps`. – Zugriffsdatum: 27. Februar 2005 74

222. **Kahan 1968** KAHAN, William M.: *A More Complete Interval Arithmetic*. Lecture Notes for a Summer Course at the University of Michigan. 17.–21. Juni 1968 148

223. **Katsirelos und Bacchus 2001** KATSIRELOS, George ; BACCHUS, Fahiem: GAC on Conjunctions of Constraints. In: WALSH, Toby (Hrsg.): *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP'01), Paphos, Zypern, 26. November – 1. Dezember 2001*. Berlin, Heidelberg, New York : Springer Verlag, 2001 (LNCS 2239), S. 610–614. – URL `http://www.cs.toronto.edu/~fbacchus/Papers/KBCP2001.pdf`. – Zugriffsdatum: 21. September 2004. – ISBN 3-540-42863-1 137

224. **Knemeyer und Schulenburg 1993** KNEMEYER, Ulrich ; SCHULENBURG, J.-Matthias Graf von der: „Expertensysteme" – Welche Faktoren fördern und hemmen die Implementation und Diffusion der Technologie in der Versicherungswirtschaft? In: (Puppe und Günter 1993), S. 248–261. – ISBN 3-540-56464-0 260

225. **Koalog 2005** Koalog Constraint Solver Tutorial / Koalog SARL. Paris, France, 2005 (Version 2.4). – Handbuch. – 23 S. – URL `http://www.koalog.com/resources/doc/jcs-tutorial.pdf`. – Zugriffsdatum: 16. März 2005 71

226. **Kolbe 2000** KOLBE, Thomas H.: *Identifikation und Rekonstruktion von Gebäuden in Luftbildern mittels unscharfer Constraints*. Aachen : Shaker Verlag, Juni 2000 (Berichte aus der Informatik 203). – iii + 149 S. – URL `http://www.ikg.uni-bonn.de/`

kolbe/publications/Dissertation/Dissertation_farbig.pdf. – Zugriffsdatum: 15. September 2004. – Zugl.: Vechta, Hochschule, Institut für Umweltwissenschaften, Dissertation, 1999. – ISBN 3-8265-7454-0 83, 84, 87, 113

227. **Krawczyk 1969** KRAWCZYK, Rudolf: Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. In: *Computing* 4 (1969), Nr. 3, S. 187–201. – ISSN 0010-485X 164

228. **Krebs 2002** KREBS, Thorsten: *Erkennen von Benutzerintentionen im inkrementellen Konfigurationsverlauf (am Beispiel von EngCon)*, Universität Bremen, Diplomarbeit, 24. April 2002. – vi + 212 S. – URL http://lki-www.informatik.uni-hamburg.de/~krebs/publications/diplom.pdf. – Zugriffsdatum: 8. Oktober 2004 21, 23

229. **Krebs et al. 2003** KREBS, Thorsten ; WAGNER, Thomas ; RUNTE, Wolfgang: Recognizing User Intentions in Incremental Configuration Processes. In: MAILHARRO, Daniel (Hrsg.): *Proceedings of the Workshop on Configuration at the 18th International Joint Conference on Artificial Intelligence (IJCAI'03).* Acapulco, Mexico, 11. August 2003, S. 44–50. – URL http://www2.ilog.com/ijcai-03/Papers/IJCAI03-08.pdf. – Zugriffsdatum: 16. September 2004 21

230. **Kühn 2000** KÜHN, Christian: Modeling Structure and Behaviour for Knowledge Based Software Configuration. In: SAUER, Jürgen (Hrsg.) ; KÖHLER, Jana (Hrsg.): *Proceedings of the 14th Workshop "New Results in Planning, Scheduling, Configuration and Design" (PuK 2000) at the 14th European Conference on Artificial Intelligence (ECAI 2000).* Humboldt-Universität zu Berlin, 21.–22. August 2000 (ECAI Workshop Notes), S. 90–97. – URL http://www-is.informatik.uni-oldenburg.de/~sauer/puk2000/papers/kuehn.pdf. – Zugriffsdatum: 8. Oktober 2004 16

231. **Kühn 2001** KÜHN, Christian: Vergleich unterschiedlicher Konfigurationsmethoden im Hinblick auf die Nutzbarkeit von Wissen über Zustandsverhalten der Konfigurationsobjekte. In: SAUER, Jürgen (Hrsg.): *Proceedings of the 15th Workshop "AI in Planning, Scheduling, Configuration and Design" (PuK'01) at the 24th Joint German/Austrian Conference on Artificial Intelligence (KI'01).* Wien, Österreich, 18. September 2001, S. 43–55. – URL http://www-is.informatik.uni-oldenburg.de/~sauer/puk2001/papers/kuehn.pdf. – Zugriffsdatum: 10. Oktober 2004 11, 13, 15, 16

232. **Kühn 2003** KÜHN, Christian: Konfigurierung eingebetteter Systeme mit Wissen über die Struktur und das Zustandsverhalten der Konfigurationsobjekte. In: HOTZ, Lothar (Hrsg.) ; KREBS, Thorsten (Hrsg.): *Beiträge zum 17. Workshop „Planen, Scheduling und Konfigurieren, Entwerfen“ (PuK'03) im Rahmen der 26. Jahrestagung Künstliche Intelligenz (KI'03).* Hamburg, 15.–16. September 2003, S. 99–108. – URL http://www-is.informatik.uni-oldenburg.de/~sauer/puk2003/paper/kuehn_puk2003.pdf. – Zugriffsdatum: 17. September 2004 16

233. **Kumar 1992** Kumar, Vipin: Algorithms for Constraints Satisfaction Problems: A Survey. In: *AI Magazine* 13 (1992), Nr. 1, S. 32–44. – URL http://www-users.cs.umn.edu/~kumar/papers/csp-aimagazine.ps. – Zugriffsdatum: 21. September 2004. – ISSN 0738-4602 83, 84, 88, 120, 129

234. **Kwan und Tsang 1996a** Kwan, Alvin C. M. ; Tsang, Edward P. K.: Minimal Forward Checking with Backmarking / University of Essex, Department of Computer Science. Wivenhoe Park, Colchester, UK, April 1996 (Technical Report CSM-260). – Forschungsbericht. – 28 S. – URL http://cswww.essex.ac.uk/CSP/papers/CSM-260.ps.Z. – Zugriffsdatum: 16. September 2004. Revidierte und gekürzte Fassung: (Kwan und Tsang 1996b) 122, 126, 368

235. **Kwan und Tsang 1996b** Kwan, Alvin C. M. ; Tsang, Edward P. K.: Minimal Forward Checking with Backmarking and Conflict-Directed Backjumping. In: Radle, Mark G. (Hrsg.): *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'96), Toulousse, France, 16.–19. November 1996.* Los Alamitos, California, USA, : IEEE Computer Society Press, 1996, S. 286–289. – URL ftp://ftp.essex.ac.uk/pub/csp/KwaTsa-ICTAI96.ps.Z. – Zugriffsdatum: 16. September 2004. – Ausführliche Version: (Kwan und Tsang 1996a). – ISBN 0-8186-7686-8 126, 368

236. **Kwan 1997** Kwan, Alvin Chi Ming: *A Framework for Mapping Constraint Satisfaction Problems to Solution Methods.* Colchester, UK, University of Essex, Department of Computer Science, PhD Thesis, Juli 1997. – xxi + 257 S. – URL ftp://ftp.essex.ac.uk/pub/csp/Kwan_PhD.ps.zip. – Zugriffsdatum: 10. November 2004 64

237. **Laudwein und Brinkop 1993** Laudwein, Norbert ; Brinkop, Axel: Konfigurieren von Rührwerken mit COMIX. In: (Puppe und Günter 1993), S. 112–124. – ISBN 3-540-56464-0 265

238. **Laveuve 1975** Laveuve, S. E.: Definition einer Kahan-Arithmetik und ihre Implementierung. In: Nickel, Karl (Hrsg.): *Interval Mathematics – Proceedings of the International Symposium, Karlsruhe, 20.–24. Mai 1975.* Berlin Heidelberg, New York : Springer Verlag, 1975 (LNCS 29), S. 236–245. – ISBN 3-540-07170-9 148, 149

239. **Lebbah und Lhomme 1998** Lebbah, Yahia ; Lhomme, Olivier: Acceleration Methods for Numeric CSPs. In: Rich, Charles (Hrsg.) ; Mostow, Jack (Hrsg.) ; Buchanan, Bruce G. (Hrsg.) ; Uthurusamy, Ramasamy (Hrsg.): *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98) and 10th Conference on Innovative Applications of Artificial Intelligence (IAAI'98), Madison, Wisconsin, USA, 26.–30. Juli 1998.* Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1998, S. 19–24. – URL http://www-sop.inria.fr/coprin/ylebbah/aaai.A297.ps. – Zugriffsdatum: 15. September 2004. – ISBN 0-262-51098-7 143, 161, 369

240. **Lebbah und Lhomme 2002** LEBBAH, Yahia ; LHOMME, Olivier: Accelerating filtering techniques for Numeric CSPs. In: *Artificial Intelligence* 139 (2002), Juli, Nr. 1, S. 109–132. – Vorhergehende Version: (Lebbah und Lhomme 1998). – ISSN 0004-3702 83, 143, 161

241. **Lesk und Schmidt 1975** LESK, Mike E. ; SCHMIDT, Eric: Lex – A Lexical Analyzer Generator / AT&T Bell Laboratories. Murray Hill, New Jersey, USA, Oktober 1975 (Computer Science Technical Report No. 39). – Forschungsbericht. – 13 S. – URL `http://dinosaur.compilertools.net/lex/lex.ps`. – Zugriffsdatum: 8. November 2005 204

242. **Lhomme 1993** LHOMME, Olivier: Consistency Techniques for Numeric CSPs. In: BAJCSY, Ruzena (Hrsg.): *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93), Chambéry, France, 28. August – 3. September 1993*. San Mateo, California, USA : Morgan Kaufmann Publishers, 1993, S. 232–238. – ISBN 1-55860-300-X 85, 143, 145, 151, 159, 160, 161, 162, 173, 212, 313

243. **Lhomme et al. 1998** LHOMME, Olivier ; GOTLIEB, Arnaud ; RUEHER, Michel: Dynamic Optimization of Interval Narrowing Algorithms. In: *The Journal of Logic Programming* 37 (1998), Oktober, Nr. 1–3, S. 165–183. – URL `http://www.essi.fr/~rueher/Publis/jlp97.pdf`. – Zugriffsdatum: 14. September 2004. – ISSN 0743-1066 160, 161

244. **Lhomme et al. 1996** LHOMME, Olivier ; GOTLIEB, Arnaud ; RUEHER, Michel ; TAILLIBERT, Patrick: Boosting the Interval Narrowing Algorithm. In: MAHER, Michael J. (Hrsg.): *Logic Programing: Proceedings of the 13th Joint International Conference and Symposium on Logic Programming (JICSLP'96), Bonn, 2.–6. September 1996*. Cambridge, Massachusetts, USA : The MIT Press, 1996, S. 378–392. – URL `http://www.essi.fr/~rueher/Publis/jicslp96.ps`. – Zugriffsdatum: 14. September 2004. – ISBN 0-262-63173-3 160, 161

245. **Liu 1996** LIU, Bing: An Improved Generic Arc Consistency Algorithm and Its Specializations. In: FOO, Norman Y. (Hrsg.) ; GOEBEL, Randy (Hrsg.): *Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence (PRICAI'96), Topics in Artificial Intelligence, Cairns, Australia, 26.–30. August 1996*. Berlin, Heidelberg, New York : Springer Verlag, 1996 (LNCS 1114), S. 264–275. – URL `http://www.cs.uic.edu/~liub/publications/pricai96.ps`. – Zugriffsdatum: 22. September 2004. – ISBN 3-540-61532-6 96

246. **Lucas 1891** LUCAS, Edouard: *Récréations Mathématiques*. 2. Aufl. Paris, France : Gauthier-Villars, 1891 114

247. **Mackworth 1977a** MACKWORTH, Alan K.: Consistency in Networks of Relations. In: *Artificial Intelligence* 8 (1977), Februar, Nr. 1, S. 99–118. – ISSN 0004-3702 57, 83, 84, 85, 90, 91, 92, 93, 94, 95, 98, 99, 102, 106, 117, 212, 309

248. **Mackworth 1977b** Mackworth, Alan K.: On Reading Sketch Maps. In: Reddy, Raj (Hrsg.): *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI'77), Cambridge, Massachusetts, USA, 22.–25. August 1977* Bd. 2. Los Altos, California, USA : William Kaufmann, 1977, S. 598–606. – ISBN 0-934613-48-6 137

249. **Mackworth 1992** Mackworth, Alan K.: The Logic of Constraint Satisfaction. In: *Artificial Intelligence* 58 (1992), Dezember, Nr. 1–3, S. 3–20. – Special Volume on Constraint Based Reasoning. – Zugl.: (Freuder und Mackworth 1994), S. 3–20. – ISSN 0004-3702 84

250. **Mackworth und Freuder 1985** Mackworth, Alan K. ; Freuder, Eugene C.: The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. In: *Artificial Intelligence* 25 (1985), Januar, Nr. 1, S. 65–74. – ISSN 0004-3702 88, 105, 135

251. **Mackworth und Freuder 1993** Mackworth, Alan K. ; Freuder, Eugene C.: The Complexity of Constraint Satisfaction Revisited. In: *Artificial Intelligence* 59 (1993), Februar, Nr. 1–2, S. 57–62. – URL `http://www.cs.ubc.ca/spider/mack/links/papers/ccsr/ccsr.pdf`. – Zugriffsdatum: 21. September 2004. – ISSN 0004-3702 84

252. **Marriott und Stuckey 1999** Marriott, Kim ; Stuckey, Peter J.: *Programming with Constraints, An Introduction.* 2. Aufl. Cambridge, Massachusetts, USA : The MIT Press, 1999. – 476 S. – ISBN 0-262-13341-5 44, 52, 53, 88, 136, 137, 142, 166, 212, 225, 226, 309, 311, 312

253. **Marti und Rueher 1995** Marti, Philippe ; Rueher, Michel: A Distributed Cooperating Constraints Solving System. In: *International Journal on Artificial Intelligence Tools (IJAIT)* 4 (1995), Juni, Nr. 1–2, S. 93–113. – URL `http://www.essi.fr/~rueher/Publis/IJAIT.ps`. – Zugriffsdatum: 12. Juni 2005. – ISSN 0218-2130 77

254. **Merlet 2002** Merlet, Jean-Pierre: ALIAS, a System Solving Library Based on Interval Analysis. In: *ERCIM News, online edition* (2002), Juli, Nr. 50. – URL `http://www.ercim.org/publication/Ercim_News/enw50/merlet.html`. – Zugriffsdatum: 11. Februar 2005 70

255. **Meyer 1995** Meyer, Manfred: *Finite Domain Constraints: Declarativity meets Efficiency, Theory meets Application.* Sankt Augustin : Infix Verlag, 1995 (DISKI 79). – 167 S. – Zugl.: Kaiserslautern, Universität, Dissertation, 1994. – ISBN 3-929037-79-3 84, 85, 86, 113

256. **Middendorf et al. 2002** Middendorf, Stefan ; Singer, Reiner ; Heid, Jörn: *Programmierhandbuch und Referenz für die Java-2-Plattform, Standard Edition.* 3., überarb. und erw. Aufl. Heidelberg : dpunkt.verlag, Oktober 2002. – 1151 S. – URL

http://www.dpunkt.de/java/index.html. – Zugriffsdatum: 7. September 2005. – ISBN 3-89864-157-0 199

257. **Minton et al. 1992** MINTON, Steven ; JOHNSTON, Mark D. ; PHILIPS, Andrew B. ; LAIRD, Philip: Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. In: *Artificial Intelligence* 58 (1992), Dezember, Nr. 1–3, S. 161–205. – Special Volume on Constraint Based Reasoning. – Zugl.: (Freuder und Mackworth 1994), S. 161–205. – ISSN 0004-3702 89, 134

258. **Mittal und Falkenhainer 1990** MITTAL, Sanjay ; FALKENHAINER, Brian: Dynamic Constraint Satisfaction Problems. In: DIETTERICH, Thomas (Hrsg.) ; SWARTOUT, William (Hrsg.): *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI'90), Boston, Massachusetts, USA, 29. Juli – 3. August 1990* Bd. 1. Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1990, S. 25–32. – ISBN 0-262-51057-X 15, 58, 59

259. **Mittal und Frayman 1989** MITTAL, Sanjay ; FRAYMAN, Felix: Towards a Generic Model of Configuration Tasks. In: (Sridharan 1989), S. 1395–1401. – ISBN 1-55860-094-9 23, 58

260. **Mohr und Henderson 1986** MOHR, Roger ; HENDERSON, Thomas C.: Arc and Path Consistency Revisited. In: *Artificial Intelligence* 28 (1986), März, Nr. 2, S. 225–233. – ISSN 0004-3702 96, 102, 106

261. **Mohr und Masini 1988** MOHR, Roger ; MASINI, Gérald: Good Old Discrete Relaxation. In: KODRATOFF, Yves (Hrsg.) ; RADIG, Bernd (Hrsg.): *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI'88), München, 1.–5. August 1988*. London, Boston : Pitman Publishing, 1988, S. 651–656. – ISBN 0-273-08798-3 91, 136, 137

262. **Monfroy 1996** MONFROY, Eric: *Solver Collaboration for Constraint Logic Programming*. France, Université Henri Poincaré – Nancy I, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine, PhD Thesis, 8. November 1996. – viii + 221 S. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/monfroy/Papers/vext_abs.ps.gz. – Zugriffsdatum: 26. Mai 2005 78

263. **Monfroy 1998** MONFROY, Eric: A Solver Collaboration in BALI. In: JAFFAR, Joxan (Hrsg.): *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'98), Manchester, UK, 15.–19. Juni 1998*. Cambridge, Massachusetts, USA : The MIT Press, 1998, S. 349–350 (Poster). – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/monfroy/Papers/poster_jicslp.ps.gz. – Zugriffsdatum: 1. Juni 2005. – ISBN 0-262-60031-5 77, 78

264. **Monfroy 2000** MONFROY, Eric: The Constraint Solver Collaboration Language of BALI. In: GABBAY, Dov (Hrsg.) ; RIJKE, Maarten de (Hrsg.): *Proceedings of the 2nd International Workshop Frontiers of Combining Systems (Fro-*

CoS'98), Amsterdam, The Netherlands, 2.–4. Oktober 1998. Baldock, Hertfordshire, UK : Research Studies Press, 2000 (Studies in Logic and Computation Vol. 7), S. 211–230. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/monfroy/Papers/frocos98.ps.gz. – Zugriffsdatum: 27. Mai 2005 77, 78, 249, 250

265. **Monfroy 2002** MONFROY, Eric: *Cooperative Constraint Solving.* France, Université de Nantes, Institut de Recherche en Informatique de Nantes, Habilitation Thesis, 22. November 2002. – viii + 212 S. – URL http://www.sciences.univ-nantes.fr/info/perso/permanents/monfroy/Papers/hdr.pdf. – Zugriffsdatum: 26. Mai 2005 78

266. **Montanari 1974** MONTANARI, Ugo: Networks of Constraints: Fundamental Properties and Applications to Picture Processing. In: *Information Sciences* 7 (1974), S. 95–132. – ISSN 0020-0255 51, 57, 83, 98, 99, 101

267. **Montanari und Rossi 1995** MONTANARI, Ugo (Hrsg.) ; ROSSI, Francesca (Hrsg.): *Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming (CP'95), Cassis, Marseilles, France, 19.–22. September 1995.* Berlin, Heidelberg, New York : Springer Verlag, 1995. (LNCS 976). – ISBN 3-540-60299-2 339, 383

268. **Moore 1969** MOORE, Ramon E.: *Intervallanalyse.* München, Wien : Oldenbourg Verlag, 1969. – 188 S. 145, 146, 147, 148, 164

269. **Nareyek 1999a** NAREYEK, Alexander: Applying Local Search to Structural Constraint Satisfaction. In: DRABBLE, Brian (Hrsg.): *Proceedings of the Workshop on Intelligent Workflow and Process Management: "The New Frontier for AI in Business" at the 16th International Joint Conference on Artificial Intelligence (IJCAI'99).* Stockholm, Sweden, 1.–2. August 1999. – URL http://www.ai-center.com/publications/nareyek-ijcai99-ws.ps. – Zugriffsdatum: 9. November 2004 62

270. **Nareyek 1999b** NAREYEK, Alexander: Structural Constraint Satisfaction. In: (Faltings et al. 1999), S. 76–82. – URL http://www.ai-center.com/publications/nareyek-aaai99-ws.ps. – Zugriffsdatum: 9. Oktober 2004. – ISBN 1-57735-089-8 62

271. **Neagu et al. 2003** NEAGU, Nicoleta ; BISTARELLI, Stefano ; FALTINGS, Boi V.: On the Computation of Local Interchangeability in Soft Constraint Satisfaction Problems. In: RUSSELL, Ingrid (Hrsg.) ; HALLER, Susan M. (Hrsg.): *Proceedings of the 16th International Florida Artificial Intelligence Research Society Conference (FLAIRS'03), Special Track on Constraint Solving and Programming, St. Augustine, Florida, USA, 12.–14. Mai 2003.* Menlo Park, California, USA : AAAI Press, 2003, S. 187–191. – URL http://www.sci.unich.it/~bista/papers/papers-download/FLAIRS03NNeagu.pdf. – Zugriffsdatum: 14. Februar 2005. – ISBN 1-57735-177-0 71

272. **Nebel 2001** Nebel, Bernhard (Hrsg.): *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01), Seattle, Washington, USA, 4.–10. August 2001.* San Mateo, California, USA : Morgan Kaufmann Publishers, 2001. – 1493 S. – ISBN 1-55860-777-3 344, 345, 386

273. **Neumann 1988** Neumann, Bernd: Configuration Expert Systems: a Case Study and Tutorial. In: Bunke, Horst O. (Hrsg.): *Artificial Intelligence in Manufacturing, Assembly and Robotics.* München, Wien : Oldenbourg Verlag, 1988, S. 27–67. – URL `http://citeseer.ist.psu.edu/neumann88configuration.html`. – Zugriffsdatum: 14. September 2004. – ISBN 3-486-20920-5 12, 17, 262, 263

274. **Neumann 1991** Neumann, Bernd: Expertensysteme zur Konstruktion: Anforderungen an ein Werkzeugsystem. In: (Cunis et al. 1991), Kap. 2, S. 12–27. – ISBN 3-540-53683-3 9, 12, 17, 261, 262

275. **Neumann et al. 1987** Neumann, Bernd ; Cunis, Roman ; Günter, Andreas ; Syska, Ingo: Wissensbasierte Planung und Konfigurierung. In: Brauer, Wilfried (Hrsg.) ; Wahlster, Wolfgang (Hrsg.): *Wissensbasierte Systeme, 2. Internationaler GI-Kongress, München, 20.–21. Oktober 1987, Proceedings.* Berlin, Heidelberg, New York : Springer Verlag, 1987 (Informatik-Fachberichte, Subreihe Künstliche Intelligenz 155), S. 347–357. – ISBN 3-540-18494-5 10

276. **Noy und McGuinness 2001** Noy, Natalya F. ; McGuinness, Deborah L.: Ontology Development 101: A Guide to Creating Your First Ontology / Stanford University. Stanford, California, USA, März 2001 (SMI Technical Report SMI-2001-0880). – Forschungsbericht. – 25 S. – URL `http://protege.stanford.edu/publications/ontology_development/ontology101.pdf`. – Zugriffsdatum: 20. September 2004 13, 23

277. **Orsvärn und Axling 1999** Orsvärn, Klas ; Axling, Tomas: The Tacton View of Configuration Tasks and Engines. In: (Faltings et al. 1999), S. 127–130. – ISBN 1-57735-089-8 18, 267

278. **Paxson 1995** Paxson, Vern: Flex, A fast scanner generator / University of California, Berkeley. Berkeley, California, USA, März 1995 (Version 2.5). – Manual. – i + 44 S. – URL `ftp://prep.ai.mit.edu/pub/gnu/Manuals/flex/ps/flex.ps.gz`. – Zugriffsdatum: 8. November 2005 204

279. **Peirce 1933** Peirce, Charles S.: Exact Logic. In: Hartshorne, Charles (Hrsg.) ; Weiss, Paul (Hrsg.): *Collected Papers of Charles Sanders Peirce* Bd. III. Cambridge, Massachusetts, USA : Harvard University Press, 1933 138

280. **Podelski 1995** Podelski, Andreas (Hrsg.): *Constraint Programming: Basics and Trends, Châtillon Spring School, Châtillon-sur-Seine, France, 16.–20. Mai 1994, Selected Papers, 22nd Spring School in Theoretical Computer Science.* Berlin, Heidelberg, New York : Springer Verlag, 1995 (LNCS 910). – xii + 316 S. – ISBN 3-540-59155-9 340, 357, 377

281. **Pollack und Georgeff 1997** POLLACK, Martha E. (Hrsg.) ; GEORGEFF, Michael P. (Hrsg.): *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Japan, 23.–29. August 1997*. San Mateo, California, USA : Morgan Kaufmann Publishers, 1997. – 1652 S. – ISBN 1-558-60480-4 344, 350, 354

282. **Pountain 1995** POUNTAIN, Dick: Constraint Logic Programming. In: *BYTE magazine*. New York, NY, USA : McGraw-Hill, Februar 1995. – URL `http://www.byte.com/art/9502/sec13/art3.htm`. – Zugriffsdatum: 16. September 2004 53

283. **Prosser 1993a** PROSSER, Patrick: Forward Checking with Backmarking / University of Strathclyde, Department of Computer Science. Glasgow, Scotland, UK, Juni 1993 (Technical Report AISL-48-93). – Forschungsbericht. – 14 S. – URL `ftp://ftp.cs.strath.ac.uk/research-reports/aisl-48-93.ps.Z`. – Zugriffsdatum: 21. September 2004. Überarb. Fassung: (Prosser 1995a) 118, 126, 374

284. **Prosser 1993b** PROSSER, Patrick: Hybrid Algorithms for the Constraint Satisfaction Problem. In: *Computational Intelligence* 9 (1993), Nr. 3, S. 268–299. – ISSN 0824-7935 77, 118, 119, 125, 126

285. **Prosser 1995a** PROSSER, Patrick: Forward Checking with Backmarking. In: MEYER, Manfred (Hrsg.): *Constraint Processing, Selected Papers*. Berlin, Heidelberg, New York : Springer Verlag, 1995 (LNCS 923), S. 185–204. – Vorhergehende Version: (Prosser 1993a). – ISBN 3-540-59479-5 118, 126, 374

286. **Prosser 1995b** PROSSER, Patrick: MAC-CBJ: Maintaining Arc Consistency with Conflict-Directed Backjumping / University of Strathclyde, Department of Computer Science. Glasgow, Scotland, UK, Mai 1995 (Research Report 95/177). – Forschungsbericht. – 7 S. – URL `http://www.dcs.st-and.ac.uk/~apes/papers/95-177.ps.Z`. – Zugriffsdatum: 16. September 2004 97, 123, 126

287. **Prosser et al. 2000** PROSSER, Patrick ; STERGIOU, Kostas ; WALSH, Toby: Singleton Consistencies. In: (Dechter 2000), S. 353–368. – URL `http://www-users.cs.york.ac.uk/~tw/Papers/pswcp00.ps`. – Zugriffsdatum: 17. September 2004. – ISBN 3-540-41053-8 110, 111

288. **Puget 1994** PUGET, Jean-François: A C++ implementation of CLP. In: *Proceedings of the 2nd Singapore International Conference on Intelligent Systems (SPICIS'94)*. Singapore, 14.–17. November 1994. – URL `http://www.ilog.com/products/optimization/tech/research/spicis94.pdf`. – Zugriffsdatum: 22. Februar 2005. – Zugl.: Technical Report 94-01. ILOG S.A. Gentilly Cedex, France, 1994 69

289. **Puget und Leconte 1995** PUGET, Jean-François ; LECONTE, Michel: Beyond the Glass Box: Constraints as Objects. In: LLOYD, John (Hrsg.): *Logic Programming, Proceedings of the 1995 International Symposium (ILPS'95), Portland, Oregon, USA,*

4.–7. Dezember 1995. Cambridge, Massachusetts, USA : The MIT Press, 1995, S. 513–527. – URL http://www.ilog.com/products/optimization/tech/research/ilps.pdf. – Zugriffsdatum: 23. Februar 2005. – ISBN 0-262-62099-5 69

290. **Puget und Van Hentenryck 1996** Puget, Jean-François ; Van Hentenryck, Pascal: A Constraint Satisfaction Approach to a Circuit Design Problem / Brown University, Computer Science Department. Providence, Rhode Island, USA, Dezember 1996 (Technical Report CS-96-34). – Forschungsbericht. – 15 S. – URL ftp://ftp.cs.brown.edu/pub/techreports/96/cs96-34.ps.Z. – Zugriffsdatum: 22. September 2004. Überarb. Fassung: (Puget und Van Hentenryck 1998) 166, 375

291. **Puget und Van Hentenryck 1998** Puget, Jean-François ; Van Hentenryck, Pascal: A Constraint Satisfaction Approach to a Circuit Design Problem. In: *Journal of Global Optimization* 13 (1998), S. 75–93. – URL http://www.cs.brown.edu/people/pvh/transistor.ps. – Zugriffsdatum: 21. September 2004. – Vorhergehende Version: (Puget und Van Hentenryck 1996). – ISSN 0925-5001 166, 375

292. **Puppe und Günter 1993** Puppe, Frank (Hrsg.) ; Günter, Andreas (Hrsg.): *Expertensysteme 93, Beiträge zur 2. Deutschen Expertensystemtagung (XPS'93), Hamburg, 17.–19. Februar 1993.* Berlin, Heidelberg, New York : Springer Verlag, 1993. (Informatik aktuell). – viii + 282 S. – ISBN 3-540-56464-0 362, 366, 368, 381, 385

293. **Purdom 1983** Purdom, Paul W.: Search Rearrangement Backtracking and Polynomial Average Time. In: *Artificial Intelligence* 21 (1983), März, Nr. 1–2, S. 117–133. – ISSN 0004-3702 129, 132

294. **Ranze et al. 2002** Ranze, Christoph ; Scholz, Thorsten ; Wagner, Thomas ; Günter, Andreas ; Herzog, Otthein ; Hollmann, Oliver ; Schlieder, Christoph ; Arlt, Volker: A Structure-Based Configuration Tool: Drive Solution Designer – DSD. In: Dechter, Rina (Hrsg.) ; Sutton, Rich (Hrsg.) ; Kearns, Michael (Hrsg.) ; Chien, Steve (Hrsg.) ; Riedl, John (Hrsg.): *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI'02) and 14th Conference on Innovative Applications of Artificial Intelligence (IAAI'02), Edmonton, Alberta, Canada, 28. Juli – 1. August 2002.* Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, September 2002, S. 845–852. – URL http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/IAAI2002.pdf. – Zugriffsdatum: 15. September 2004. – ISBN 0-262-51129-0 1, 21, 22, 28, 34, 36, 69, 272, 277

295. **Ringwelski 2001a** Ringwelski, Georg: Distributed Constraint Satisfaction with Cooperating Asynchronous Solvers. In: Walsh, Toby (Hrsg.): *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP'01), Paphos, Zypern, 26. November – 1. Dezember 2001.* Berlin, Heidelberg, New York : Springer Verlag, 2001 (LNCS 2239), S. 777 (Poster). –

URL http://4c.ucc.ie/web/upload/publications/inProc/cp01doc-proc.pdf. – Zugriffsdatum: 19. Januar 2005. – ISBN 3-540-42863-1 64

296. **Ringwelski 2001b** RINGWELSKI, Georg: A New Execution Model for Constraint Processing in Object-Oriented Software. In: *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP'01), Kiel, 13.–15. September 2001*, Universität Kiel (veröffentlicht als technischer Bericht Nr. 2017), 2001. – URL http://4c.ucc.ie/web/upload/publications/inProc/wflp01.pdf. – Zugriffsdatum: 19. Februar 2005 71

297. **Ringwelski 2002** RINGWELSKI, Georg: Object-Oriented Constraint Programming with J.CP. In: COELLO COELLO, Carlos A. (Hrsg.) ; ALBORNOZ, Alvaro de (Hrsg.) ; SUCAR, Luis E. (Hrsg.) ; BATTISTUTTI, Osvaldo C. (Hrsg.): *Advances in Artificial Intelligence, Proceedings of the 2nd Mexican International Conference on Artificial Intelligence (MICAI'2002), Merida, Yucatan, Mexico, 22.–26. April 2002*. Berlin, Heidelberg, New York : Springer Verlag, 2002 (LNCS 2313), S. 194–203. – URL http://4c.ucc.ie/web/upload/publications/inProc/micai02.pdf. – Zugriffsdatum: 19. Januar 2005. – ISBN 3-540-43475-5 69, 71

298. **Ringwelski 2003** RINGWELSKI, Georg: *Asynchrones Constraintlösen – Ein generisches Ausführungsmodell zur adaptiven, inkrementellen Constraintverarbeitung.* Berlin, Technische Universität, Fakultät IV – Elektrotechnik und Informatik, Dissertation, 12. Februar 2003. – viii + 177 S. – URL http://4c.ucc.ie/web/upload/publications/phdThesis/diss.pdf. – Zugriffsdatum: 19. Januar 2005 64, 69, 71, 72, 75, 79

299. **Ringwelski und Schlenker 2002** RINGWELSKI, Georg ; SCHLENKER, Hans: Dynamic Distributed Constraint Satisfaction with Asynchronous Solvers. In: APT, Krzysztof R. (Hrsg.) ; FAGES, François (Hrsg.) ; FREUDER, Eugene C. (Hrsg.) ; O'SULLIVAN, Barry (Hrsg.) ; ROSSI, Francesca (Hrsg.) ; WALSH, Toby (Hrsg.): *Proceedings of the Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP'02).* Cork, Ireland, 19.–21. Juni 2002, S. 211–220. – URL http://4c.ucc.ie/web/upload/publications/inProc/ercim02acs.pdf. – Zugriffsdatum: 19. Januar 2005 64

300. **Rossi et al. 1989** ROSSI, Francesca ; PETRIE, Charles J. ; DHAR, Vasant: On the Equivalence of Constraint Satisfaction Problems / Microelectronics and Computer Technology Corporation. Austin, Texas, USA, Dezember 1989 (MCC Technical Report ACT-AI-222-89). – Forschungsbericht. – 29 S. – URL http://snrc.stanford.edu/~petrie/constraint-def.pdf. – Zugriffsdatum: 20. September 2004. Revidierte und gekürzte Fassung: (Rossi et al. 1990) 138, 140, 377

301. **Rossi et al. 1990** ROSSI, Francesca ; PETRIE, Charles J. ; DHAR, Vasant: On the Equivalence of Constraint Satisfaction Problems. In: AIELLO, Luigia Carlucci (Hrsg.): *Proceedings of the 9th European Conference on Artificial Intelligence*

(ECAI'90), Stockholm, Sweden, 6.–10. August 1990. London, Boston : Pitman Publishing, 6.–10. August 1990, S. 550–556. – Ausführliche Version: (Rossi et al. 1989). – ISBN 0-273-08822-X 140, 376

302. **Roy et al. 1999** Roy, Pierre ; Liret, Anne ; Pachet, François: Constraint Satisfaction Problems Framework. In: Fayad, Mohamed E. (Hrsg.) ; Schmidt, Douglas C. (Hrsg.) ; Johnson, Ralph E. (Hrsg.): *Implementing Application Frameworks: Object-Oriented Frameworks at Work.* Chichester, London, New York : John Wiley & Sons, September 1999 (Wiley Computer Publishing), Kap. 17, S. 369–401. – URL `http://www.csl.sony.fr/downloads/papers/1997/roy97a.pdf`. – Zugriffsdatum: 23. Februar 2005. – ISBN 0-471-25201-8 74

303. **Roy et al. 2000** Roy, Pierre ; Liret, Anne ; Pachet, François: The Framework Approach for Constraint Satisfaction. In: *ACM Computing Surveys (CSUR)* 32 (2000), März, Nr. 1es. – URL `http://www.csl.sony.fr/downloads/papers/1998/roy98a.pdf`. – Zugriffsdatum: 22. Februar 2005. – CSUR Electronic Symposium on Object-Oriented Application Frameworks – Artikel Nr. 13. – ISSN 0360-0300 74, 178

304. **Roy und Pachet 1997** Roy, Pierre ; Pachet, François: Reifying Constraint Satisfaction in Smalltalk. In: *Journal of Object-Oriented Programming (JOOP)* 10 (1997), Juli/August, Nr. 4, S. 43–51. – URL `http://www-poleia.lip6.fr/~fdp/MyPapers/BackTalk/BackTalk-joop.ps.Z`. – Zugriffsdatum: 25. Februar 2005 74

305. **Roy et al. 1997** Roy, Pierre ; Pachet, François ; Perrot, Jean-François: A Framework for Expressing Knowledge about Constraint Satisfaction Problems. In: Dankel II, Douglas D. (Hrsg.): *Proceedings of the 10th International Florida Artificial Intelligence Research Symposium Conference (FLAIRS'97).* Daytona Beach, Florida, USA, 12.–14. Mai 1997, S. 47–51. – URL `http://citeseer.ist.psu.edu/roy97framework.html`. – Zugriffsdatum: 25. Februar 2005 74

306. **Rueher 1995** Rueher, Michel: An Architecture for Cooperating Constraint Solvers on Reals. In: (Podelski 1995), S. 231–250. – URL `http://www.essi.fr/~rueher/Publis/SpringSchool.ps`. – Zugriffsdatum: 12. Juni 2005. – Zugl.: Research Report 94-54, Informatique, Signaux et Systèmes de Sophia Antipolis (I3S), Université de Nice Sophia Antipolis, France, 1994. – ISBN 3-540-59155-9 77

307. **Russel und Norvig 2002** Russel, Stuart J. ; Norvig, Peter: *Artificial Intelligence: A Modern Approach (The Intelligent Agent Book).* 2. Aufl. Upper Saddle River, New Jersey, USA : Prentice Hall, Dezember 2002 (Prentice Hall Series in Artificial Intelligence). – 1132 S. – URL `http://aima.cs.berkeley.edu/newchap05.pdf`. – Zugriffsdatum: 7. September 2005. – ISBN 0-13790-395-2 59, 83, 89, 120, 129, 134, 144, 157, 166, 225, 226

308. **Ruttkay 1994** Ruttkay, Zsófia: Fuzzy Constraint Satisfaction. In: *Proceedings of the Third IEEE International Conference on Fuzzy Systems at the IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, 26.–29. Juni 1994.*

New York, NY, USA : IEEE Neural Networks Council, September 1994, S. 1263–1268. – URL http://www.cwi.nl/~zsofi/Publications/IEEE94.ps. – Zugriffsdatum: 28. Oktober 2004. – ISBN 0-78031-896-X 61, 62

309. **Ruttkay 1998** RUTTKAY, Zsófia: Constraint Satisfaction – a Survey. In: *CWI Quarterly* 11 (1998), Juni/September, Nr. 2–3, S. 123–161. – URL ftp://ftp.cwi.nl/pub/CWIQuarterly/1998/11.2-3/ruttkay.pdf. – Zugriffsdatum: 21. September 2004. – ISSN 0168-826X 83, 128, 133, 143

310. **Sabin und Freuder 1994a** SABIN, Daniel ; FREUDER, Eugene C.: Contradicting Conventional Wisdom in Constraint Satisfaction. In: BORNING, Alan (Hrsg.): *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94), Rosario, Orcas Island, Washington, USA, 2.–4. Mai 1994*. Berlin, Heidelberg, New York : Springer Verlag, 1994 (LNCS 874), S. 10–20. – URL http://4c.ucc.ie/web/upload/publications/inProc/sabin94contradicting.pdf. – Zugriffsdatum: 21. Januar 2006. – Zugl.: (Sabin und Freuder 1994b). – ISBN 3-540-58601-6 122, 123, 125, 126, 378

311. **Sabin und Freuder 1994b** SABIN, Daniel ; FREUDER, Eugene C.: Contradicting Conventional Wisdom in Constraint Satisfaction. In: COHN, Anthony G. (Hrsg.): *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI'94), Amsterdam, The Netherlands, 8.–12. August 1994*. Chichester, London, New York : John Wiley & Sons, 1994, S. 125–129. – URL http://citeseer.ist.psu.edu/186864.html. – Zugriffsdatum: 20. Januar 2006. – Zugl.: (Sabin und Freuder 1994a). – ISBN 0-471-95069-6 123, 126, 378

312. **Sabin und Freuder 1996a** SABIN, Daniel ; FREUDER, Eugene C.: Configuration as Composite Constraint Satisfaction. In: FALTINGS, Boi V. (Hrsg.) ; FREUDER, Eugene C. (Hrsg.): *Configuration – Papers from the AAAI Fall Symposium, Technical Report FS-96-03, Boston, Massachusetts, USA, 9.–11. November 1996*. Menlo Park, California, USA : AAAI Press, 1996, S. 28–36. – URL http://4c.ucc.ie/web/upload/publications/inProc/sabin96configuration.pdf. – Zugriffsdatum: 20. Oktober 2004. – Zugl.: (Sabin und Freuder 1996b). – ISBN 1-57735-018-9 12, 59, 378

313. **Sabin und Freuder 1996b** SABIN, Daniel ; FREUDER, Eugene C.: Configuration as Composite Constraint Satisfaction. In: LUGER, George F. (Hrsg.): *Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop*. Menlo Park, California, USA : AAAI Press, 1996, S. 153–161. – URL http://citeseer.ist.psu.edu/sabin96configuration.html. – Zugriffsdatum: 20. Oktober 2004. – Zugl.: (Sabin und Freuder 1996a). – ISBN 1-57735-003-0 12, 59, 378

314. **Sabin und Freuder 1997** SABIN, Daniel ; FREUDER, Eugene C.: Understanding and Improving the MAC Algorithm. In: SMOLKA, Gert (Hrsg.): *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97), Schloss Hagenberg, Linz, Austria, 29. Oktober – 1.*

November 1997. Berlin, Heidelberg, New York : Springer Verlag, 1997 (LNCS 1330), S. 167–181. – URL http://4c.ucc.ie/web/upload/publications/inProc/sabin97understanding.ps. – Zugriffsdatum: 16. September 2004. – ISBN 3-540-63753-2 125

315. **Sabin und Weigel 1998** Sabin, Daniel ; Weigel, Rainer: Product Configuration Frameworks – A Survey. In: *IEEE Intelligent Systems* 13 (1998), Juli/August, Nr. 4, S. 42–49. – ISSN 1094-7167 1, 11, 12, 15, 269

316. **Sabin und Freuder 1998** Sabin, Mihaela ; Freuder, Eugene C.: Detecting and Resolving Inconsistency and Redundancy in Conditional Constraint Satisfaction Problems. In: *Proceedings of the Workshop on Constraint Problem Reformulation at the 4th International Conference on Principles and Practice of Constraint Programming (CP'98)*. Pisa, Italy, 30. Oktober 1998. – URL http://ic-www.arc.nasa.gov/people/frank/sabin.cp98.FINAL.ps. – Zugriffsdatum: 19. Oktober 2004. – Zugl.: (Sabin und Freuder 1999) 59, 379

317. **Sabin und Freuder 1999** Sabin, Mihaela ; Freuder, Eugene C.: Detecting and Resolving Inconsistency and Redundancy in Conditional Constraint Satisfaction Problems. In: (Faltings et al. 1999), S. 90–94. – URL http://www.rivier.edu/faculty/msabin/web/home/publications/cp98work.ps. – Zugriffsdatum: 19. Oktober 2004. – Zugl.: (Sabin und Freuder 1998). – ISBN 1-57735-089-8 59, 379

318. **Sam-Haroud und Faltings 1996a** Sam-Haroud, Djamila ; Faltings, Boi V.: Consistency Techniques for Continuous Constraints. In: *Constraints, An International Journal* 1 (1996), September, Nr. 1–2, S. 85–118. – URL http://liawww.epfl.ch/Publications/Archive/Sam-Haroud1996a.pdf. – Zugriffsdatum: 14. September 2004. – ISSN 1383-7133 143, 168, 169, 170

319. **Sam-Haroud und Faltings 1996b** Sam-Haroud, Djamila ; Faltings, Boi V.: Solving Non-Binary Convex CSPs in Continuous Domains. In: (Freuder 1996), S. 410–424. – URL http://liawww.epfl.ch/Publications/Archive/Sam-Haroud1996.pdf. – Zugriffsdatum: 14. September 2004. – ISBN 3-540-61551-2 143, 169, 170

320. **Sam-Haroud 1995** Sam-Haroud, Jamila: *Constraint Consistency Techniques for Continuous Constraints*. Lausanne, Switzerland, Swiss Federal Institute of Technology (EPFL), PhD Thesis No. 1423, 1995. – xviii + 178 S. – URL http://liawww.epfl.ch/~haroud/Thesis-SamHaroud.ps.gz. – Zugriffsdatum: 14. September 2004 83, 143, 144, 153, 163, 167

321. **Schiex et al. 1996** Schiex, Thomas ; Régin, Jean-Charles ; Gaspin, Christine ; Verfaillie, Gérard: Lazy Arc Consistency. In: (Clancey et al. 1996), S. 216–221. – URL http://www.inra.fr/bia/T/schiex/Export/lac-aaai96.ps.gz. – Zugriffsdatum: 15. September 2004. – ISBN 0-262-51091-X 108

322. **Schlenker und Ringwelski 2003** Schlenker, Hans ; Ringwelski, Georg: POOC: A Platform for Object-Oriented Constraint Programming. In: O'Sullivan,

Barry (Hrsg.): *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP'02), Cork, Ireland, 19.–21. Juni 2002, Selected Papers*. Berlin, Heidelberg, New York : Springer Verlag, 2003 (LNCS 2627), S. 159–170. – URL `http://4c.ucc.ie/web/upload/publications/inProc/ercim02pooc.pdf`. – Zugriffsdatum: 13. Januar 2005. – ISBN 3-540-00986-8 68, 75

323. **Schlingheider 1994** SCHLINGHEIDER, Jörg ; SPUR, Günter (Hrsg.): *Methodik zur Entwicklung rechnergestützter Konfigurationssysteme*. München, Wien : Carl Hanser Verlag, 1994 (Produktionstechnik – Berlin 145). – 156 S. – Zugl.: Berlin, Technische Universität, Dissertation, 1994. – ISBN 3-446-17827-9 20

324. **Schnelle 2004** SCHNELLE, Dirk: Kurze Einführung in log4j / The Apache Software Foundation. Forest Hill, Maryland, USA, 8. Juni 2004. – Handbuch. – URL `http://www.javacore.de/tutorials/schnelle/log4jmanual.pdf`. – Zugriffsdatum: 7. November 2005. Vorhergehende Version: (Gülcü 2002) 200, 360

325. **Schöning 2000** SCHÖNING, Uwe: *Logik für Informatiker*. 5. Aufl. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2000. – 200 S. – ISBN 3-8274-1005-3 101

326. **Schwalb und Vila 1998** SCHWALB, Eddie ; VILA, Lluís: Temporal Constraints: A Survey. In: *Constraints, An International Journal* 3 (1998), Juni, Nr. 2–3, S. 129–149. – URL `http://www.ics.uci.edu/~csp/r66.pdf`. – Zugriffsdatum: 18. Oktober 2004. – ISSN 1383-7133 58

327. **SICStus 2004** SICStus Prolog User's Manual / Swedish Institute of Computer Science, Intelligent Systems Laboratory. Kista, Sweden, November 2004 (Release 3.12.0). – Handbuch. – xviii + 834 S. – URL `http://www.sics.se/sicstus/docs/3.12.0/pdf/sicstus.pdf`. – Zugriffsdatum: 20. Januar 2005 66, 75

328. **Silaghi et al. 1999** SILAGHI, Marius-Călin ; SAM-HAROUD, Djamila ; FALTINGS, Boi V.: Intelligent Domain Splitting for CSPs with Ordered Domains. In: (Jaffar 1999), S. 488–489 (Poster). – URL `http://www.cs.fit.edu/~msilaghi/papers/CP99poster.pdf`. – Zugriffsdatum: 14. September 2004. – ISBN 3-540-66626-5 151

329. **Silaghi et al. 2001** SILAGHI, Marius-Călin ; SAM-HAROUD, Djamila ; FALTINGS, Boi V.: Search Techniques for Non-Linear Constraints with Inequalities. In: STROULIA, Eleni (Hrsg.) ; MATWIN, Stan (Hrsg.): *Advances in Artificial Intelligence, Proceedings of the 14th Biennial Conference of the Canadian Society for Computational Studies of Intelligence (AI'01), Ottawa, Canada, 7.–9. Juni 2001*. Berlin, Heidelberg, New York : Springer Verlag, 2001 (LNCS 2056), S. 183–193. – URL `http://www.cs.fit.edu/~msilaghi/papers/AI2001.pdf`. – Zugriffsdatum: 17. September 2004. – ISBN 3-540-42144-0 143, 166, 170, 171

330. **Sillito 2000** SILLITO, Jonathan: *Arc Consistency for General Constraint Satisfaction Problems and Estimating the Cost of Solving Constraint Satisfaction Problems*. Edmonton, Alberta, Canada, University of Alberta, Master's Thesis, 2000. – 73 S.

– URL `http://ai.uwaterloo.ca/~vanbeek/publications/sillito.ps.gz`. – Zugriffsdatum: 21. September 2004 137

331. **Simon 1993** SIMON, K.-H.: Wo sind sie, all die Expertensysteme? Einstieg in ein Monitoring im Bereich der Umweltanwendungen. In: (Puppe und Günter 1993), S. 262–268. – ISBN 3-540-56464-0 11, 260

332. **Singh 1995** SINGH, Moninder: Path Consistency Revisited. In: VASSILOPOULOS, John F. (Hrsg.): *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'95), Herndon, Virginia, USA, 5.–8. November 1995.* Los Alamitos, California, USA, : IEEE Computer Society Press, 1995, S. 318–325. – URL `ftp://ftp.cis.upenn.edu/pub/msingh/ictai95.ps.Z`. – Zugriffsdatum: 15. September 2004. – ISBN 0-8186-7312-5 102, 106

333. **Smith 1992** SMITH, Barbara M.: How to Solve the Zebra Problem, or Path Consistency the Easy Way. In: NEUMANN, Bernd (Hrsg.): *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92), Vienna, Austria, 3.–7. August 1992.* Chichester, London, New York : John Wiley & Sons, 1992, S. 36–37. – ISBN 0-471-93608-1 125

334. **Smith et al. 2000** SMITH, Barbara M. ; STERGIOU, Kostas ; WALSH, Toby: Using Auxiliary Variables and Implied Constraints to Model Non-Binary Problems. In: KAUTZ, Henry (Hrsg.) ; PORTER, Bruce (Hrsg.) ; ENGELMORE, Robert (Hrsg.) ; HIRSH, Haym (Hrsg.): *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000) and 12th Conference on Innovative Applications of Artificial Intelligence (IAAI 2000), Austin, Texas, USA, 30. Juli – 3. August 2000.* Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 2000, S. 182–187. – URL `http://www-users.cs.york.ac.uk/~tw/Papers/sswaaai00.pdf`. – Zugriffsdatum: 21. September 2004. – ISBN 0-262-51112-6 135, 142

335. **Sridharan 1989** SRIDHARAN, Natesa S. (Hrsg.): *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI'89), Detroit, Michigan, USA, 20.–26. August 1989.* San Mateo, California, USA : Morgan Kaufmann Publishers, 1989. – ISBN 1-55860-094-9 351, 355, 363, 371

336. **Stearns 1997** STEARNS, Beth: Java Native Interface / Sun Microsystems. Santa Clara, California, USA, 19. Februar 1997. – The Java Tutorial: A practical guide for programmers. – URL `http://java.sun.com/docs/books/tutorial/native1.1/index.html`. – Zugriffsdatum: 7. November 2005. Zugl.: (Stearns 1998) 68, 73, 220, 381

337. **Stearns 1998** STEARNS, Beth: Java Native Interface. In: CAMPIONE, Mary (Hrsg.) ; WALRATH, Kathy (Hrsg.) ; HUML, Alison (Hrsg.): *The Java Tutorial Continued: The Rest of the JDK.* Reading, Massachusetts, USA : Addison-Wesley, Dezember 1998 (The Java Series from the Source), S. 625–680. – Zugl.: (Stearns 1997). – ISBN 0-201-48558-3 68, 73, 220, 381

338. **Stergiou und Walsh 1999** Stergiou, Kostas ; Walsh, Toby: Encodings of Non-Binary Constraint Satisfaction Problems. In: Hendler, James (Hrsg.) ; Subramanian, Devika (Hrsg.) ; Uthurusamy, Ramasamy (Hrsg.) ; Hayes-Roth, Barbara (Hrsg.): *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99) and 11th Conference on Innovative Applications of Artificial Intelligence (IAAI'99), Orlando, Florida, USA, 18.–22. Juli 1999*. Menlo Park, California, USA/-Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, 1999, S. 163–168. – URL http://dream.dai.ed.ac.uk/group/tw/papers/swaaai99.ps. – Zugriffsdatum: 15. September 2004. – ISBN 0-262-51106-1 140, 141, 142

339. **Stumptner 1997** Stumptner, Markus: An Overview of Knowledge-Based Configuration. In: *AI Communications (AICOM)* 10 (1997), Juli, Nr. 2, S. 111–125. – ISSN 0921-7126 1, 11, 12, 58

340. **Stumptner et al. 1998** Stumptner, Markus ; Friedrich, Gerhard E. ; Haselböck, Alois: Generative Constraint-Based Configuration of Large Technical Systems. In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)* 12 (1998), September, Nr. 4, S. 307–320. – Special Issue: Configuration Design. – ISSN 0890-0604 15, 59

341. **Stumptner und Haselböck 1993** Stumptner, Markus ; Haselböck, Alois: A Generative Constraint Formalism for Configuration Problems. In: Torasso, Pietro (Hrsg.): *Advances in Artificial Intelligence, Proceedings of the 3rd Congress of the Italian Association for Artificial Intelligence (AI*IA'93), Torino, Italy, 26.–28. Oktober 1993*. Berlin, Heidelberg, New York : Springer Verlag, 1993 (LNCS 728), S. 302–313. – ISBN 3-540-57292-9 59

342. **Sutschet 2001** Sutschet, Gerhard: *COMIX – System zur Konfigurierung von Maschinen und Anlagen*. Karlsruhe : Fraunhofer-Institut für Informations- und Datenverarbeitung (IITB), November 2001. – URL http://www.iitb.fraunhofer.de/servlet/is/624/Comix.pdf?command=downloadContent&filename=Comix.pdf. – Zugriffsdatum: 27. August 2004 17, 265

343. **Syska 1991** Syska, Ingo: Modulare Expertensystemarchitekturen. In: (Cunis et al. 1991), Kap. 3, S. 28–36. – ISBN 3-540-53683-3 260

344. **Syska und Cunis 1991** Syska, Ingo ; Cunis, Roman: Constraints in PLAKON. In: (Cunis et al. 1991), Kap. 6, S. 77–91. – ISBN 3-540-53683-3 1, 41, 42, 50, 60

345. **Torrens 1997** Torrens, Marc: *An Application using the Java Constraint Library: The Air Travel Planning System*. Lausanne, Switzerland, Swiss Federal Institute of Technology (EPFL), Artificial Intelligence Laboratory, Diploma Thesis, 27. Juni 1997. – x + 65 S. – URL http://www.marctorrens.com/downloads/diploma/diploma.ps. – Zugriffsdatum: 14. Februar 2005 70

346. **Torrens et al. 1997a** Torrens, Marc ; Weigel, Rainer ; Faltings, Boi V.: Java Constraint Library. In: Frühwirth, Thom (Hrsg.) ; Hermengildo, Manuel

(Hrsg.) ; TARAU, Paul (Hrsg.) ; CODOGNET, Philippe (Hrsg.) ; ROSSI, Francesca (Hrsg.): *Proceedings of the Workshop on Constraint Reasoning on the Internet at the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97).* Schloss Hagenberg, Linz, Austria, 1. November 1997. – URL http://liawww.epfl.ch/Publications/Archive/Torrens1997a.pdf. – Zugriffsdatum: 14. Februar 2005. – Vorhergehende Version: (Torrens et al. 1997b) 71, 383

347. **Torrens et al. 1997b** TORRENS, Marc ; WEIGEL, Rainer ; FALTINGS, Boi V.: Java Constraint Library: bringing constraints technology on the Internet using the Java language. In: FREUDER, Eugene C. (Hrsg.): *Constraints & Agents – Papers from the AAAI Workshop at the 14th National Conference on Artificial Intelligence (AAAI'97), Providence, Rhode Island, USA, 27.–31. Juli 1997, Technical Report WS-97-05.* Menlo Park, California, USA : AAAI Press, 1997. – URL http://liawww.epfl.ch/Publications/Archive/Torrens1997.pdf. – Zugriffsdatum: 14. Februar 2005. – Erweiterte Fassung: (Torrens et al. 1997a). – ISBN 1-57735-032-4 71, 383

348. **Torrens et al. 1998** TORRENS, Marc ; WEIGEL, Rainer ; FALTINGS, Boi V.: Distributing Problem Solving on the Web Using Constraint Technology. In: *Proceedings of the 10th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'98), Taipei, Taiwan, 10.–12. November 1998.* Los Alamitos, California, USA, : IEEE Computer Society Press, 1998, S. 42–49. – URL http://liawww.epfl.ch/Publications/Archive/Torrens1998.pdf. – Zugriffsdatum: 14. Februar 2005. – ISBN 0-7803-5214-9 71

349. **Tsang 1993** TSANG, Edward P. K.: *Foundations of Constraint Satisfaction.* London, San Diego, New York : Academic Press, 1993 (Computation in Cognitive Science). – 421 S. – URL http://cswww.essex.ac.uk/CSP/papers/Tsang-Fcs1993.pdf/. – Zugriffsdatum: 20. September 2004. – ISBN 0-12-701610-4 52, 54, 56, 83, 84, 85, 87, 89, 101, 105, 107, 109, 112, 116, 118, 120, 122, 124, 128, 129, 130, 131, 132, 134

350. **Tsang 1998** TSANG, Edward P. K.: No more "Partial" and "Full Looking Ahead". In: *Artificial Intelligence* 98 (1998), Januar, Nr. 1–2, S. 351–361. – ISSN 0004-3702 109, 123

351. **Van Hentenryck 1989** VAN HENTENRYCK, Pascal: *Constraint Satisfaction in Logic Programming.* Cambridge, Massachusetts, USA : The MIT Press, 1989. – xvi + 224 S. – ISBN 0-262-08181-4 83, 84, 85, 136

352. **Van Hentenryck 1995** VAN HENTENRYCK, Pascal: Constraint Solving for Combinatorial Search Problems: A Tutorial. In: (Montanari und Rossi 1995), S. 564–587. – ISBN 3-540-60299-2 66, 67

353. **Van Hentenryck et al. 1992** VAN HENTENRYCK, Pascal ; DEVILLE, Yves ; TENG, Choh-Man: A Generic Arc-Consistency Algorithm and its Specializations. In: *Artificial Intelligence* 57 (1992), Oktober, Nr. 2–3, S. 291–321. – ISSN 0004-3702 96, 106

354. **Van Hentenryck et al. 1995** Van Hentenryck, Pascal ; McAllester, David A. ; Kapur, Deepak: Solving Polynomial Systems Using a Branch and Prune Approach / Brown University, Computer Science Department. Providence, Rhode Island, USA, Februar 1995 (Technical Report CS-95-01). – Forschungsbericht. – URL `ftp://ftp.cs.brown.edu/pub/techreports/95/cs95-01.ps.Z`. – Zugriffsdatum: 21. September 2004. Überarb. Fassung: (Van Hentenryck et al. 1997) 165, 384

355. **Van Hentenryck et al. 1997** Van Hentenryck, Pascal ; McAllester, David A. ; Kapur, Deepak: Solving Polynomial Systems Using a Branch and Prune Approach. In: *SIAM Journal on Numerical Analysis* 34 (1997), April, Nr. 2, S. 797–827. – URL `http://www.cs.unm.edu/~kapur/mypapers/siam.97.ps.gz`. – Zugriffsdatum: 21. September 2004. – Vorhergehende Version: (Van Hentenryck et al. 1995). – ISSN 0036-1429 151, 165, 384

356. **Van Hentenryck und Saraswat 1996** Van Hentenryck, Pascal ; Saraswat, Vijay: Strategic Directions in Constraint Programming. In: *ACM Computing Surveys (CSUR)* 28 (1996), Dezember, Nr. 4, S. 701–726. – Special ACM 50th-Anniversary Issue: Strategic Directions in Computing Research. – Zugl.: Constraints, 2 (1997), Nr. 1, S. 7–33. – ISSN 0360-0300 62, 63, 67, 89

357. **Vu et al. 2003a** Vu, Xuan-Ha ; Sam-Haroud, Djamila ; Faltings, Boi V.: Clustering for Disconnected Solution Sets of Numerical CSPs. In: Apt, Krzysztof R. (Hrsg.) ; Fages, François (Hrsg.) ; Rossi, Francesca (Hrsg.) ; Szeredi, Péter (Hrsg.) ; Váncza, József (Hrsg.): *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP'03), Budapest, Hungary, 30. Juni – 2. Juli 2003, Selected Papers*. Berlin, Heidelberg, New York : Springer Verlag, 2003 (LNCS 3010), S. 25–43. – URL `http://liawww.epfl.ch/Publications/Archive/vuxuanha2003a.pdf`. – Zugriffsdatum: 17. September 2004. – ISBN 3-540-21834-3 151

358. **Vu et al. 2003b** Vu, Xuan-Ha ; Sam-Haroud, Djamila ; Faltings, Boi V.: Clustering the Search Tree for Numerical Constraints. In: *Notes of the 2nd International Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS'03)*. Lausanne, Switzerland, 18.–21. November 2003. – URL `http://liawww.epfl.ch/Publications/Archive/vuxuanha2003b.pdf`. – Zugriffsdatum: 17. September 2004 151

359. **Vu et al. 2002** Vu, Xuan-Ha ; Sam-Haroud, Djamila ; Silaghi, Marius-Călin: Approximation Techniques for Non-linear Problems with Continuum of Solutions. In: Koenig Sven, Robert C. (Hrsg.): *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation (SARA'02), Kananaskis, Alberta, Canada, 2.–4. August 2002*. Berlin, Heidelberg, New York : Springer Verlag, 2002 (LNCS 2371), S. 224–241. – URL `http://liawww.epfl.ch/Publications/Archive/vuxuanha2002b.pdf`. – Zugriffsdatum: 17. September 2004. – ISBN 3-540-43941-2 143, 166

360. **Vu et al. 2003c** VU, Xuan-Ha ; SAM-HAROUD, Djamila ; SILAGHI, Marius-Călin: Numerical Constraint Satisfaction Problems with Non-Isolated Solutions. In: BLIEK, Christian (Hrsg.) ; JERMANN, Christophe (Hrsg.) ; NEUMAIER, Arnold (Hrsg.): *Global Optimization and Constraint Satisfaction, Proceedings of the 1st International Workshop on Global Constraint Optimization and Constraint Satisfaction (COCOS'02), Valbonne-Sophia Antipolis, France, 2.–4. Oktober 2002, Revised Selected Papers*. Berlin, Heidelberg, New York : Springer Verlag, 2003 (LNCS 2861), S. 194–210. – URL `http://liawww.epfl.ch/Publications/Archive/vuxuanha2002c.pdf`. – Zugriffsdatum: 17. September 2004. – ISBN 3-540-20463-6 143, 166, 170, 171

361. **Wallace 1993** WALLACE, Richard J.: Why AC-3 is Almost Always Better than AC-4 for Establishing Arc Consistency in CSPs. In: BAJCSY, Ruzena (Hrsg.): *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93), Chambéry, France, 28. August – 3. September 1993*. San Mateo, California, USA : Morgan Kaufmann Publishers, 1993, S. 239–245. – ISBN 1-55860-300-X 96, 171

362. **Waltz 1972** WALTZ, David L.: Generating Semantic Descriptions from Drawings of Scenes with Shadows / Massachusetts Institute of Technology. Cambridge, Massachusetts, USA, 1972 (Technical Report AI-TR-271). – Forschungsbericht 85

363. **Waltz 1975** WALTZ, David L.: Understanding Line Drawings of Scenes with Shadows. In: WINSTON, Patric Henry (Hrsg.): *The Psychology of Computer Vision*. New York, NY, USA : McGraw-Hill, 1975, S. 19–91. – ISBN 0-07-071048-1 52, 57, 83, 84, 85, 91, 95

364. **Werres 2002** WERRES, York: *Effizientes und flexibles Pattern-Matching für komplexe Objektstrukturen*, Universität Bremen, Diplomarbeit, 21. April 2002. – 129 S. 21, 35

365. **Weule 1993** WEULE, Hartmut: Expertensysteme im industriellen Einsatz. In: (Puppe und Günter 1993), S. 1–12. – ISBN 3-540-56464-0 9, 260

366. **Wilson und Borning 1993** WILSON, Molly ; BORNING, Alan: Hierarchical Constraint Logic Programming. In: *The Journal of Logic Programming* 16 (1993), Juli/August, Nr. 3–4, S. 227–318. – URL `ftp://ftp.cs.washington.edu/tr/1993/01/UW-CSE-93-01-02a.PS.Z`. – Zugriffsdatum: 1. November 2004. – Special Issue on Constraint Logic Programming. – ISSN 0743-1066 62

367. **Yang und Yang 1997** YANG, Christopher C. ; YANG, Ming-Hsuan: Constraint Networks: A Survey. In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Orlando, Florida, USA, 12.–15. Oktober 1997* Bd. 2, Institute of Electrical and Electronics Engineers (IEEE), November 1997, S. 1930–1935. – URL `http://vision.ai.uiuc.edu/mhyang/papers/csp-survey.ps.gz`. – Zugriffsdatum: 14. September 2004. – ISBN 0-780-34053-1 83, 143

368. **Yokoo und Hirayama 2000** YOKOO, Makoto ; HIRAYAMA, Katsutoshi: Algorithms for Distributed Constraint Satisfaction: A Review. In: *Autonomous*

Agents and Multi-Agent Systems 3 (2000), Juni, Nr. 2, S. 185–207. – URL http://www.kecl.ntt.co.jp/csl/ccrg/members/yokoo/PDF/aams2000.pdf. – Zugriffsdatum: 9. November 2004. – ISSN 1387-2532 63

369. **Yu und Skovgaard 1998** YU, Bei ; SKOVGAARD, Jørgen: A Configuration Tool to Increase Product Competitiveness. In: *IEEE Intelligent Systems* 13 (1998), Juli/August, Nr. 4, S. 34–40. – ISSN 1094-7167 17, 266

370. **Zhang und Yap 2001** ZHANG, Yuanlin ; YAP, Roland H. C.: Making AC-3 an Optimal Algorithm. In: (Nebel 2001), S. 316–321. – URL http://www.dcs.gla.ac.uk/~pat/cp4/papers/ac2001b.pdf. – Zugriffsdatum: 16. September 2004. – ISBN 1-55860-777-3 97, 106

371. **Zhou 1999** ZHOU, Neng-Fa: DJ: Declarative Java User's Manual / Kyushu Institute of Technology, Faculty of Computer Science and Systems Engineering. Fukuoka, Japan, 1999 (Version 0.5). – Handbuch. – iii + 48 S. – URL http://www.cad.mse.kyutech.ac.jp/people/zhou/dj/manual/manual.html. – Zugriffsdatum: 18. März 2005 71

372. **Zhou et al. 1998** ZHOU, Neng-Fa ; KANEKO, Sousuke ; YAMAUCHI, Kouji: DJ: A Java-based Constraint Language and System. In: *Proceedings of the 15th Annual Japan Society for Software Science and Technology Conference (JSSST'98)*. Japan, 1998. – URL http://www.cad.mse.kyutech.ac.jp/people/zhou/papers/jssst98.ps.gz. – Zugriffsdatum: 18. März 2005 71

373. **Zoeteweij 2002** ZOETEWEIJ, Peter: Coordination-Based Solver Cooperation in DICE. In: GRANVILLIERS, Laurent (Hrsg.): *Proceedings of the Workshop on Cooperative Solvers in Constraint Programming (CoSolv'02) at the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*. Ithaca, New York, USA, 8. September 2002. – URL http://homepages.cwi.nl/~peterz/papers/zoeteweij-cosolv02.pdf. – Zugriffsdatum: 25. Juni 2005 78

374. **Zoeteweij 2003** ZOETEWEIJ, Peter: A Coordination-Based Framework for Distributed Constraint Solving. In: O'SULLIVAN, Barry (Hrsg.): *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP'02), Cork, Ireland, 19.–21. Juni 2002, Selected Papers*. Berlin, Heidelberg, New York : Springer Verlag, 2003 (LNCS 2627), S. 171–184. – URL http://homepages.cwi.nl/~peterz/papers/zoeteweij-ercim02.pdf. – Zugriffsdatum: 25. Juni 2005. – ISBN 3-540-00986-8 78, 79

Abkürzungsverzeichnis

Allgemeine Abkürzungen

aktual.	aktualisiert
Anm.	Anmerkung
Aufl.	Auflage
Bd.	Band
bzgl.	bezüglich
bspw.	beispielsweise
bzw.	beziehungsweise
ca.	circa
engl.	englisch
erw.	erweiterte
et al.	et alii
etc.	et cetera
evtl.	eventuell
f.	folgend
ff.	fortfolgend
gdw.	genau dann wenn
ggf.	gegebenenfalls
Hrsg.	Herausgeber
inkl.	inklusive
d. h.	das heißt
i. A.	im Allgemeinen
i. d. R.	in der Regel
Kap.	Kapitel
max.	maximal
min.	minimal/mindestens
Nr.	Nummer
o. ä.	oder ähnlich
S.	Seite
sog.	sogenannt
u.	und
u. a.	und andere

überarb.	überarbeitet
usf.	und so fort
usw.	und so weiter
u. U.	unter Umständen
vgl.	vergleiche
vollst.	vollständig
vs.	versus
z. B.	zum Beispiel
zit.	zitiert
z. T.	zum Teil
zugl.	zugleich

Fachbegriffliche Abkürzungen

AC	*arc consistency*
ACS	Asynchronous Constraint Solving
ACSP	Adaptive Constraint Satisfaction Problem
AI	Artificial Intelligence
API	Application Programming Interface
BC	Backchecking
BJ	Backjumping
BM	Backmarking
BNF	Backus-Naur-Form
BSD	Berkeley Source Distribution
BT	(chronologisches) Backtracking
CAL	Contrainte Avec Logique
CBJ	konfliktbasiertes Backjumping
CCP	Concurrent Constraint Programming
CCSP	Continuous Constraint Satisfaction Problem
CHIP	Constraint Handling in Prolog
CHR	Constraint Handling Rules
CIAL	Constraint Interval Arithmetic Language
CLOS	Common Lisp Object System
CLP	Constraint Logic Programming
CLP(BNR)	Constraint Logic Programming with Booleans, Naturals and Reals
CLP(FD)	Constraint Logic Programming with Finite Domains
CLP(Intervals)	Constraint Logic Programming with Intervals
CLP(R)	Constraint Logic Programming with Reals
CompCSP	Composite Constraint Satisfaction Problem
CondCSP	Conditional Constraint Satisfaction Problem
CORBA	Common Object Request Broker Architecture
CP	Constraint Programming

CSP	Constraint Satisfaction Problem
CSPDL	CSP Description Language
CWA	*closed world assumption*
DAC	*directional arc consistency*
DCSP	Dynamic Constraint Satisfaction Problem
DDBT	Dependency Directed Backtracking
DisCSP	Distributed Constraint Satisfaction Problem
DJ	Declarative Java
DOM	Document Object Model
DPC	*directional path consistency*
DSR	*dynamic search rearrangement*
DTD	Document Type Definition
DVO	*dynamic variable ordering*
EJB	Enterprise Java Beans
FAQ	Frequently Asked Questions
FC	Forward Checking
FD	*finite domain*
FF	*fail first*
FLA	Full Look-Ahead
GBJ	graphenbasiertes Backjumping
GCC	GNU Compiler Collection
GPL	GNU General Public License
GCSP	Generative Constraint Satisfaction Problem
GDCC	Guarded Definite Clauses with Constraints
GT	Generate & Test
GUI	Graphical User Interface
HCLP	Hierarchical Constraint Logic Programming
HCSP	Hierarchical Constraint Satisfaction Problem
IA	Intervallarithmetik
ICSP	Interval Constraint Satisfaction Problem
IDE	Integrated Development Environment
HTML	Hypertext Markup Language
J2EE	Java 2 Enterprise Edition
J2SE	Java 2 Standard Edition
JAR	Java Archiv
JAXP	Java API for XML Processing
JCL	Java Constraint Library
JNI	Java Native Interface
JSP	Java Server Pages
KCS	Koalog Constraint Solver
KI	Künstliche Intelligenz
KS-Fokus	Konfigurierungsschritt-Fokus
LAC	*lazy arc consistency*
LFC	Lazy Forward Checking

LGPL	GNU Lesser General Public License
LP	*logic programming*
LVO	*look-ahead value ordering*
MAC	*maintaining arc consistency*
MAS	Multi-Agenten-System
MaxCSP	Maximal Constraint Satisfaction Problem
MBO	*minimal bandwidth ordering*
MC	*min-conflicts*
MCO	*maximum cardinality ordering*
MD	*max-domain*
MDO	*maximum degree ordering*
MFC	Minimal Forward Checking
Mixed CSP	Mixed Constraint Satisfaction Problem
MRV	*minimum remaining values*
MWO	*minimal width ordering*
NC	*node consistency*
NCSP	Numerical Constraint Satisfaction Problem
NIC	*neighborhood inverse consistency*
ORB	Object Request Broker
OCL	Object Constraint Language
OCS	Over-Constrained System
OCSP	Open Constraint Satisfaction Problem
ODBC	Open DataBase Connectivity
OOP	objektorientierte Programmierung
OPS5	Official Production System, Version 5
PC	*path consistency*
PCSP	Partial Constraint Satisfaction Problem
PDS	*point-domain-size*
PIC	*path inverse consistency*
PLA	Partial Look-Ahead
RPC	*restricted path consistency*
SAC	*singleton arc consistency*
SAT	Satisfiability Problem
SCSP	Structural Constraint Satisfaction Problem
SDK	Software Development Kit
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
SoftCSP	Soft Constraint Satisfaction Problem
SQL	Structured Query Language
SRPC	*singleton restricted path consistency*
SVO	*static variable ordering*
TCSP	Temporal Constraint Satisfaction Problem
TMS	Truth Maintenance System
TP	Toleranzpropagation

UML	Unified Modeling Language
URL	Uniform Resource Locator
WMD	*weighted-max-domain*
XML	Extended Markup Language
Yacc	Yet Another Compiler-Compiler
YACS	Yet Another Constraint Solver
YCM	YACS Constraint-Manager

Organisationen, Institute und Unternehmen

AAAI	American Association for Artificial Intelligence
ACM	Association for Computing Machinery
AFCET	Association Française pour la Cybernétique Economique et Technique
AiS	GMD-Institut für Autonome intelligente Systeme
ALP	Association for Logic Programming
APES	Algorithms, Problems and Empirical Studies Research Group
ASI	NATO Advanced Study Institute
CEDAR	Center of Excellence for Document Analysis and Recognition
CNRS	Centre National de la Recherche Scientifique
CoLogNET	Computational Logic Network
COPRIN	Constraints Solving, Optimisation, Robust Interval Analysis
CWI	Centrum voor Wiskunde en Informatica
DEC	Digital Equipment Corporation
EAPLS	European Association for Programming Languages and Systems
ECRC	European Computer-Industry Research Centre
EPFL	Ecole Polytechnique Fédérale de Lausanne
ERCIM	European Research Consortium for Informatics and Mathematics
FIRST	GMD-Institut für Rechnerarchitektur und Softwaretechnik
FIT	GMD-Institut für Angewandte Informationstechnik
FLAIRS	bis 1999: Florida Artificial Intelligence Research Symposium, ab 2000: Florida Artificial Intelligence Research Society
GI	Gesellschaft für Informatik
GDR	Groupe de Recherche
GMD	Gesellschaft für Mathematik und Datenverarbeitung
GNU	GNU's Not UNIX
I3S	Informatique, Signaux et Systèmes de Sophia Antipolis
ICS	Information and Computer Science Department
IEEE	Institute of Electrical and Electronics Engineers
INRIA	Institut National de Recherche en Informatique et en Automatique
IITB	Fraunhofer-Institut für Informations- und Datenverarbeitung
JSSST	Japan Society for Software Science and Technology

LIRMM	Le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
MIT	Massachusetts Institute of Technology
NVTI	Nederlandse Vereniging voor Theoretische Informatica
OMG	Object Management Group
PLANSIG	UK Planning and Scheduling Special Interest Group
SICS	Swedish Institute of Computer Science
SIGAPP	ACM Special Interest Group on Applied Computing
SIGPLAN	ACM Special Interest Group on Programming Languages
TKT	GMD-Institut für Telekooperationstechnik
TZI	Technologie-Zentrum Informatik
UCI	University of California, Irvine
W3C	World Wide Web Consortium

Konferenzen und Workshops

AISMC	Conference on Artificial Intelligence and Symbolic Mathematical Computation
CAIA	Conference on Artificial Intelligence for Applications
COCOS	Workshop on Global Constraint Optimization and Constraint Satisfaction
CONCUR	Conference on Concurrency Theory
CoSolv	Workshop on Cooperative Solvers in Constraint Programming
COTIC	Workshop on Concurrent Constraint Programming for Time Critical Applications and Multi-Agent Systems
CP	Conference on Principles and Practice of Constraint Programming (ab 1995, vorher: PPCP)
CPDC	Workshop on Constraint Programming for Decision and Control
CSCLP	Workshop on Constraint Solving and Constraint Logic Programming
ECAI	European Conference on Artificial Intelligence
EPIA	Portuguese Conference on Artificial Intelligence
FGCS	Fifth Generation Computer Systems
FLAIRS	bis 1999: Florida Artificial Intelligence Research Symposium Conference, ab 2000: Florida Artificial Intelligence Research Society Conference
FroCoS	Workshop Frontiers of Combining Systems
GWAI	German Workshop on Artificial Intelligence
IAAI	Conference on Innovative Applications of Artificial Intelligence
ICLP	International Conference on Logic Programming
ICTAI	IEEE International Conference on Tools with Artificial Intelligence
IJCAI	International Joint Conference on Artificial Intelligence
ILPS	International Logic Programming Symposium

INAP	International Conference on Applications of Prolog
ISWC	International Semantic Web Conference
JFPLC	Journées Francophones de Programmation en Logique et Programmation par Contraintes
JICSLP	Joint International Conference and Symposium on Logic Programming
JSSST	Japan Society for Software Science and Technology Conference
KI	Deutsche Jahrestagung Künstliche Intelligenz
KR	Conference on Principles of Knowledge Representation and Reasoning
LPSS	Logic Programming Summer School
MultiCPL	Workshop on Multiparadigm Constraint Programming Languages
OCS	Workshop on Over-Constrained Systems
OOPSLA	ACM Conference on Object-Oriented Programming Systems, Languages and Applications
PACLP	Conference and Exhibition on Practical Application of Constraint Technologies and Logic Programming
PADL	Workshop on Practical Aspects of Declarative Languages
PAIS	Conference on Prestigious Applications of Intelligent Systems
PA Java	Conference and Exhibition on Practical Application of Java
PPCP	Conference on Principles and Practice of Constraint Programming (bis 1994, danach: CP)
PRICAI	Pacific Rim International Conference on Artificial Intelligence
PuK	GI Workshop Planen, Scheduling und Konfigurieren, Entwerfen
RFIA	Reconnaissance des Formes et Intelligence Artificielle
SAC	ACM Symposium on Applied Computing
SARA	Symposium on Abstraction, Reformulation and Approximation
SCAI	Scandinavian Conference Conference on Artificial Intelligence
SPICIS	Singapore International Conference on Intelligent Systems
TRICS	Workshop on Techniques foR Implementing Constraint programming Systems
WDS	Week of Doctoral Students
WFLP	Workshop on Functional and (Constraint) Logic Programming
XPS	Deutsche Expertensystemtagung

Zeitschriften, Serien und Enzyklopädien

AI	Artificial Intelligence
AICOM	AI Communications
AI EDAM	Artificial Intelligence for Engineering Design, Analysis and Manufacturing
CACM	Communications of the ACM
CSUR	ACM Computing Surveys

DISKI	Dissertationen zur Künstlichen Intelligenzw
ECS	Encyclopedia of Cognitive Science
ENTCS	Electronic Notes in Theoretical Computer Science
IJAIT	International Journal on Artificial Intelligence Tools
JACM	Journal of the ACM
JAIR	Journal of Artificial Intelligence Research
JFLP	Journal of Functional and Logic Programming
LNCS	Lecture Notes in Computer Science
MITECS	The MIT Encyclopedia of the Cognitive Sciences
PAMI	IEEE Transactions on Pattern Analysis and Machine Intelligence
TOCHI	ACM Transactions on Computer-Human Interaction

Stichwortverzeichnis

D

R

S

T

Kolophon

Stell dir vor, es geht, und keiner kriegt's hin.

WOLFGANG NEUSS

Diese Arbeit wäre in der vorliegenden Form ohne die Satzbeschreibungssprache TeX von *Donald E. Knuth*[1] und dem darauf basierenden Makropaket LaTeX von *Leslie Lamport*[2] nicht möglich gewesen.

Der Text wurde mit LaTeX 2_ε aus der 10 Punkt Computer-Modern-Schrift gesetzt. Für den Schriftsatz fanden die ec-Schriften[3] von *Jörg Knappen* Verwendung.

Die Abbildungen in dieser Arbeit wurden mittels Dia[4], xfig[5], OpenOffice.org Draw[6], Kontour[7], Inkscape[8] und Together Developer[9] erstellt.

Das Literaturverzeichnis wurde nach DIN 1505 mit BibTeX und dem dinat-Paket[10] von *Helge Baumann* sowie dem natbib-Paket[11] von *Patrick W. Daly* erzeugt.

Die Miniaturbildchen („Thumbnails") für die PDF-Version dieses Dokuments wurden mit dem thumbpdf-Paket[12] von *Heiko Oberdiek* generiert und eingebunden.

Der Satz dieses Dokuments erfolgte am 26. Januar 2006 um 19:06 Uhr auf einer unter Debian GNU/Linux[13] („Sid", unstable) betriebenen Intel Celeron-Maschine („Tualatin") mit 1.4 Gigahertz Taktung unter Verwendung der teTeX-Distribution in der Version 2.0.2.

Zur vollständigen Auflösung aller Referenzen benötigte LaTeX insgesamt sechs Läufe.

Alle Eingabedateien brachten eine Summe von 11889 Kilobytes auf. Die Größe der Ausgabedateien betrug im DVI-Format 1773 Kilobytes, im PS-Format 13486 Kilobytes, im PDF-Format 6086 Kilobytes und im HTML-Format 4371 Kilobytes.

---

[1] `http://www-cs-faculty.stanford.edu/~knuth/`
[2] `http://research.microsoft.com/users/lamport/`
[3] `http://www.uni-mainz.de/~knappen/jk007.html`
[4] `http://www.gnome.org/projects/dia/`
[5] `http://www.xfig.org`
[6] `http://www.openoffice.org`
[7] `http://www.koffice.org`
[8] `http://www.inkscape.org`
[9] `http://www.borland.com`
[10] `ftp://ftp.dante.de/tex-archive/biblio/bibtex/contrib/german/dinat/`
[11] `ftp://ftp.dante.de/tex-archive/macros/latex/contrib/natbib/`
[12] `ftp://ftp.dante.de/tex-archive/support/thumbpdf/`
[13] `http://www.debian.org`