Asynchrones Constraintlösen

Ein generisches Ausführungsmodell zur adaptiven, inkrementellen Constraintverarbeitung

vorgelegt durch
Diplom-Informatiker
Georg Ringwelski
aus Berlin

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
– Dr. ing. –

genehmigte Dissertation
Promotionsausschuß:
Vorsitzender: Prof. Dr. Hermann Krallmann
Berichter: Prof. Dr. Stefan Jähnichen
Berichter:Prof. Dr. Dr. Thom Frühwirth

Tag der wissenschaftlichen Aussprache: 12.02.2003

Berlin 2003
D 83

Kurzfassung

Constraint satisfaction Probleme (CSP) bestehen aus einer Menge von Randbedingungen (Constraints), die einen gesuchten Lösungszustand beschreiben, in dem alle Constraints erfüllt sein müssen. Las Lösen von CSP ist ein aktuelles Forschungsgebiet mit vielen Anwendungsfeldern in Wirtschaft, Technik und dem alltäglichen Leben. Die Erstellung von Terminkalendern zum Beispiel ist ein solches Problem, dessen Randbedingungen durch persönliche Präferenzen und durch probleminhärente Anforderungen, wie z.B. die Überschneidungsfreiheit der Termine, gegeben sind. Eine Lösung dieses CSP besteht darin, Terminkalender für alle beteiligten Personen herzustellen, so daß möglichst alle Termine wahrgenommen werden können. CSP sind im Allgemeinen NP-vollständig, so daß in der Künstlichen Intelligenz nach Verfahren gesucht wird, die trotz des erforderlichen Nichtdeterminismus schnell genug für ihr jeweiliges Anwendungsgebiet eine oder mehrere Lösungen finden.

Zur Lösung von CSP hat sich die Verarbeitung der Constraints aus Modellen in einer bestimmten Constraintdomäne mit Hilfe entsprechender Constraintlöser durchgesetzt. Solche Systeme generieren aus der domänenspezifischen Problembeschreibung eine Lösung, indem sie einen generischen Algorithmus auf das konkrete Modell anwenden. Planungsprobleme werden zum Beispiel oft durch Relationen in den ganzen Zahlen modelliert und durch *finite domain* Constraintlöser gelöst. Dazu werden Constraintvariablen genau die Werte zugeordnet, die alle Randbedingungen des Modells erfüllen. In dynamischen Umgebungen können sich die Modelle, z.B. durch Benutzerinteraktion, beliebig ändern. Hier werden dynamische Löser benötigt, die die Lösung inkrementell an die veränderten Modelle anpassen, möglichst ohne alle Berechnungen erneut auszuführen.

Das Hinzufügen neuer Constraints zu einem gegebenen Modell wird in inkrementellen Constraintlösern durch Propagation umgesetzt. Das neue Wissen wird sofort so weitgehend wie möglich, bzw. zeitlich vertretbar, mit dem Alten verknüpft um weiteres Wissen abzuleiten. Propagation kann synchron oder auch asynchron durchgeführt werden, so daß dieses erfolgreiche Konzept auch in nebenläufigen bzw. verteilten Systemen anwendbar ist. Wenn ein Constraint aus dem Modell entfernt wird, muß sein Effekt und damit all seine Konsequenzen rückgängig gemacht werden. Diese Operation wird in den meisten constraintverarbeitenden Systemen aber nicht unterstützt, weil der Propagationsprozeß im Nachhinein nur dann nachvollziehbar und umkehrbar ist, wenn der Vorgang beobachtet und "abgespeichert" wurde. Für die Constraintrücknahme in synchronen Systemen sind einige Algorithmen bekannt und in prototypischen Systemen implementiert. Allerdings setzten diese Algorithmen einen sequentiellen Programmablauf (synchrone Propagation) voraus, so daß sie in nebenläufigen Systemen nicht anwendbar sind.

In dieser Arbeit wird ein Ausführungsmodell für einen generischen dynamischen Constraintlöser definiert, der auch nebenläufige Interaktion und paralleles Lösen von CSP in kooperierenden Constraintlösern unterstützt. Das Ausführungsmodell ist für beliebige Constraintdomänen anwendbar, weil spezialisierte Constraints durch entsprechende Komponenten leicht integrierbar sind. Constraints werden durch Propagationsmethoden definiert, die Einschränkungen an Variablendomänen in Abhängigkeit anderer Variablen vornehmen. In einer prototypischen Anwendung zur Terminplanung, die im Laufe der Arbeit als Fallstudie vorgestellt wird, werden zum Beispiel Constraints implementiert, die die

möglichen Anfangs- und Endzeitpunkte von Terminen so einschränken, daß es keine Überschneidungen gibt.

Mit der Theorie der Chaotischen Iteration wurde Ende der 90er Jahre ein theoretischer Rahmen zur Formalisierung der Propagation von Constraints geschaffen. Das zentrale Ergebnis dieser Arbeiten war, daß Propagation auch in chaotischer Ausführung zu einem eindeutigen Fixpunkt führt. In der vorliegenden Arbeit wird der Fixpunkt der Chaotischen Iteration als denotationale Semantik von Constraintprogrammen angenommen. Die Domänenreduktionsfunktionen, mit denen der Fixpunkt theoretisch berechnet wird können eindeutig aus den Implementierungen der Constraints abgeleitet werden. Um diese Semantik zu berechnen wird in dieser Dissertation ein praktisch anwendbares Ausführungsmodell für imperative Constraintprogramme definiert. Im Gegensatz zur Chaotischen Iteration ist diese Berechnung nachweislich endlich und es können während der Propagation Funktionen mit Seiteneffekten und Constraintrücknahme ausgeführt werden. Diese zusätzlichen Möglichkeiten erlauben neben der Bearbeitung nebenläufiger dynamischer CSP auch die Integration von nichtdeterministischen Suchalgorithmen in den Propagationsprozeß als Constraints.

Nichtdeterministische Suche ist zum Lösen von CSP notwendig, weil diese in der Regel NP-vollständig sind. Im vorgestellen Ausführungsmodell sind beliebige Suchalgorithmen als Constraints integrierbar, wobei das nichtdeterministische "Ausprobieren" von Werten des Algorithmus durch Absetzen und Zurücknehmen von Instantiierungs-Constraints umgesetzt wird.

Weil die Propagation chaotisch und asynchron ausgeführt wird, kann das Ausführungsmodell zur Lösung von asynchron auftretende Probleme verwendet werden. Die Eingabe erfolgt dabei durch (gegebenenfalls nebenläufige) Constraintprogramme, die in kooperierenden Constraintlösern beliebig Constraints hinzufügen und zurücknehmen können. Als Ausgabe kann das aktuelle Ergebnis immer in den Variablen abgelesen werden. Durch diese Funktionalität erschließen sich neue Anwendungsgebiete in dynamischen und verteilten Umgebungen, die bisher für generische Problemlöser mit diesem effizienten Lösungsalgorithmus nicht zugänglich waren. Das offene, verteilte System zur Terminplanung aus der Fallstudie zum Beispiel, bei dem an kontinuierlichen Agenten nebenläufig Constraints abgesetzt und zurückgenommen werden können, wäre mit herkömmlichen Constraintlösern nicht umzusetzen.

Danksagung

Mein Dank gilt vor allem Herrn Prof. Dr. Stefan Jähnichen, dem Leiter unseres Forschungsinstituts Fraunhofer FIRST, für die Betreuung dieser Dissertation. Herrn Prof. Dr. Dr. Thom Frühwirth möchte ich für die Begutachtung dieser Arbeit und für seine Unterstützung im Umgang mit der internationalen Forschungsgemeinschaft danken. Herrn Prof. Dr. Ulrich Geske, dem Leiter unserer Forschungsgruppe, und Herrn Dr. Armin Wolf, dem Forschungskoordinator des Instituts aus unserer Gruppe, möchte ich besonders danken. Sie haben mich mit ihrem Interesse an wissenschaftlicher Arbeit während meiner Forschungen immer gut beraten und unterstützt. Durch ihre Anerkennung meiner Forschungsleistung bekam ich den nötigen Freiraum, diese Arbeit zu erstellen. Auch den Initiatoren des Doktorandenprogramms der ehemaligen GMD, in dessen Rahmen ich ideale Arbeitsbedingungen vorfand, möchte ich danken.

Die freundschaftliche Atmosphäre in unserer Forschungsgruppe unter Herrn Geskes Leitung hat mir die Möglichkeit zu vielen anregenden, wissenschaftlichen Gesprächen und Diskussionen gegeben. All meine Kollegen und besonders Hans Schlenker und Armin Wolf hatten immer ein offenes Ohr für meine Fragen und konnten mir oft wichtige Hinweise, Anregungen und Hilfen für meine Forschung geben. Bei der Erstellung dieser Arbeit waren mir Hans Schlenker, Ulrich Geske, Armin Wolf und Ole Boysen mit vielfältigen Anregungen zum Inhalt der einzelnen Kapitel sehr hilfreich. Außerdem hat Kirsten Schröder, inhaltlichem Unverständnis trotzend, Stil und Rechtschreibung der Arbeit für mich durchgesehen. Für diese Unterstützung bei der Vorbereitung und der Erstellung des vorliegenden Textes möchte ich mich herzlich bedanken.

Besonders unterstützt hat mich meine Frau Yvonne, die, vor allem in den letzten Monaten, die Folgen meiner Belastung geduldig ertragen hat und sicherstellte, daß mein Leben außerhalb des Forschungsinstituts so schön weiterging wie bisher.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der Einleitung der vorliegenden Dissertationsschrift wird die Relevanz constraintbasierter Methoden als Modellierungs- und Programmierparadigma dargestellt und begründet. Das Zusammenwirken von domänenspezifischen Constraints und Domänenunabhängiger Constraintverarbeitung wird erläutert. Zur Motivation dieser Arbeit werden einige neue Anwendungsbereiche der Constraintprogrammierung angegeben aus denen sich Anforderungen an das neue Ausführungsmodell ableiten lassen. Diese Anforderungen werden als angestrebte Ergebnisse dieser Arbeit identifiziert. Schließlich wird eine Einführung in die durchgängige Fallstudie des Textes gegeben.

Constraintbasierte Methoden haben sich als Modellierungs- und Programmierparadigma in vielen Anwendungsbereichen der Informatik durchgesetzt. So werden zum Beispiel in den Bereichen Künstliche Intelligenz, Datenbanken, Kombinatorische Optimierung oder Erstellung von Benutzerschnittstellen constraintbasierte Programme verwendet. Besonders zur Kombinatorischen Optimierung hat sich Constraintprogrammierung als sehr geeignetes Paradigma bei der Erstellung von Software für Planung und Konfiguration sowohl im akademischen als auch im industriellen Bereich erwiesen.

Der Erfolg der Constraintprogrammierung ergibt sich aus ihrer Deklarativität: Zur Modellierung werden Constraints (die Relationen repräsentieren) über Variablen (die Objekte repräsentieren) angegeben. Die Integration dieser Modellierung in Programmiersprachen, die die Constraints verarbeiten, ermöglicht die Generierung von Problemlösungen aus dem deklarativen Modell des Problems selbst. Constraintprogrammierung beruht also auf der Trennung eines abstrakten, wiederverwendbaren Constraintlösers von deklarativen Programmen, die konkrete Probleme modellieren. Die Anpassung der Lösung an Veränderungen der Problemstellung ergibt sich direkt durch die Modifikation der Modelle; indem Constraints hinzugefügt oder entfernt werden.

Die effizienten Constraint-Programmiersysteme, mit denen bekannte *benchmark*-Probleme oft vergleichsweise schnell gelöst werden können, sind ein weiterer Grund für den Erfolg der Constraintprogrammierung. Solche Programmiersysteme können als Bibliotheken für bekannte Programmiersprachen (z.B. Java), als Erweiterung einer bestimmten Sprache (z.B. Prolog) oder als eigenständige Sprachen (z.B. Mozart) realisiert werden. Unabhängig von der Umsetzung kann man in allen Systemen Constraints und Constraintverarbeitung differenzieren.

Constraints sind jeweils spezifisch für einen Anwendungsbereich, die Constraintdomäne, aus dem die Werte für die Variablen gewählt werden. Die möglichen Werte, die immer Teilmengen der Constraintdomäne sind, werden dabei für die Variablen explizit in Variablendomänen oder implizit durch einstellige Constraints verwaltet. Einige bekannte Constraintdomänen, für die effiziente Constraint-Programmiersysteme zur Verfügung stehen, sind z.B. endliche Mengen ganzer Zahlen [Hen89], endliche Mengen [Ger97], Bäume [Col84] oder Intervalle reeller Zahlen [JMSY92]. Die wesentliche Eigenschaft von Constraints ist Propagation. Durch sie werden Werte aus den Variablendomänen entfernt, die das Constraint nicht erfüllen. Welche Werte das sind, wird im Constraint aus dem implementierten Wissen über die Constraintdomäne und ggf. aus den Domänen andere Variablen des Constraints abgeleitet.

Constraintverarbeitung ist dagegen unabhängig von Constraintdomänen. Sie wird mit dem Ausführungsmodell des Programmiersystems bestimmt und gibt an, wie Constraints verarbeitet werden [Sch02]. Die Erzeugung, das Absetzen und die Rücknahme von Constraints sowie das Zusammenwirken unterschiedlicher Constraints wird durch das Ausführungsmodell festgelegt. Es wird z.B. angegeben, wann die Propagation eines Constraints durchgeführt wird oder wie die Variablendomänen bei der Constraintrücknahme berechnet werden. Ein besonders wichtiger Bereich der Constraintverarbeitung ist Suche. Weil viele Probleme durch Propagation alleine nicht gelöst werden können, muß ein Suchalgorithmus, der nichtdeterministisch nach Lösungen sucht, integriert werden[MS98]. Der Suchalgorithmus prüft verschiedene Kombinationen von Variablenbelegungen auf ihre Konsistenz bzgl. der Propagation aller abgesetzten Constraints. Constraintprobleme werden normalerweise erst dann als gelöst betrachtet, wenn eine konsistente Belegung für alle Variablen gefunden wurde.

1.1 Motivation und Ziele

Zusammenfassung. Das erfolgreiche Konzept der Constraintprogrammierung ermöglicht die effiziente Lösung vieler Probleme aus der Künstlichen Intelligenz. Neue Anwendungsfelder für das Konzept entstehen durch verteilte oder dynamische Constraintprobleme, die jeweils Gegenstand aktueller Forschung sind. Ein Constraintlöser oder Ausführungsmodell, das verteilte und dynamische Probleme verarbeitet ist aber bisher nicht bekannt.

Die Effizienz constraintbasierter Verfahren zum Lösen klassischer meist NP-vollständiger Probleme, z.B. aus der Künstlichen Intelligenz, resultiert aus dem angewandten Lösungsalgorithmus. Durch die Propagation der Constraints können die Variablendomänen vor und während der Anwendung des eigentlichen Suchalgorithmus oft so weit eingeschränkt werden, daß sich der Aufwand der nichtdeterministischen Suche auf ein Maß reduziert, das die Behandlung mancher Probleme erst erlaubt. Beim bekannten *send+more=money*-Problem [GW] zum Beispiel existieren a priori 10^8 verschiedene Variablenbelegungen, die untersucht werden müssen, um darunter die einzige Lösung zu finden. Durch die Ausführung der Constraints ohne Suche, z.B. in der constraintlogischen Programmiersprache SICStus [SICS], reduziert sich diese Zahl auf $2 * 10^7$. Wird dann während der nichtdeterministischen Suche z.B. der Wert 9 für die Variable s gewählt, reduziert sich der Suchraum sofort auf $4 * 3 * 7^3$, also auf ca. 0.4% der theoretischen Anzahl, der zu untersuchende Variablenbelegungen.

Seit den 80er Jahren werden viele professionelle und akademische Constraintlöser entwickelt, die ihre Anwendung in Software-Produkten bzw. in der Erforschung weiterer Möglichkeiten der Constraintprogrammierung finden. Anfangs wurden sehr spezialisierte Systeme, wie z.B. [Wal75], entwickelt, um Constraintprogrammierung durch Effizienz und Deklarativität für verschiedene Anwendungsbereiche attraktiv zu machen. Mit der Zeit entwickelten sich dann allgemeinere Systeme, die jeweils die effiziente Verarbeitung von Constraints einer bestimmten Constraintdomäne erlauben, z.B. [Chip, JMSY92]. Constraint-basierte Programme für diese Systeme erfordern eine domänenspezifische Art der Modellierung, nämlich eine Abbildung des realen Problems in Funktionen und Relationen der jeweiligen Constraintdomäne. In der gegenwärtigen Forschung wird u.a. versucht, die effizienten Methoden der verschiedenen Systeme zur Constraintverarbeitung zusammenzufügen, um die Menge der verarbeitbaren Modelle zu vergrößern [Mon96, Hof01]. Eine solche Kooperation von Constraintlösern ermöglicht es, spezielle und damit meist effizientere Methoden zur Lösung von Teilproblemen zu verwenden. Dies muß allerdings mit dem Aufwand der Koordination, die die Kooperation der Teillösungen steuert, bezahlt werden. Ein generisches Ausführungsmodell zu definieren, mit dem man effiziente Algorithmen für konkrete Probleme individuell zusammenstellen kann, ohne dadurch erheblichen Koordinationsaufwand zu erzeugen, kann damit als eine lohnende Aufgabe angesehen werden.

Aber nicht nur bezüglich der Constraintdomänen bzw. der Constraints erweiterten sich die Anwendungsfelder der Constraintprogrammierung, sondern auch bezüglich der Constraintverarbeitung. Die nebenläufige oder verteilte Ausführung von Berechnungen zum Lösen von Constraintproblemen zum Beispiel ist ein wichtiges Forschungsgebiet im Bereich Constraintprogrammierung geworden [YD98, SSHF00, Han01, MR99]. Wenn parallele Constraintverarbeitung möglich ist, können Probleme gelöst werden, die zu groß sind, um in monolithischen, deterministischen Systemen bearbeitet zu werden oder die aus Gründen des Datenschutz (*privacy*) oder der Sicherheit (*security*) nicht zentralisiert werden dürfen. Diese neuen Anwendungsfelder zur Constraintverarbeitung bringen neue Probleme mit sich, für deren Lösung die herkömmlichen Ausführungsmodelle nur bedingt und mit großem Aufwand einsetzbar sind.

Auch die dynamische Verarbeitung von Constraints, d.h. das dynamische Hinzufügen und Entfernen von Constraints während der Propagation oder Suche, ist seit Mitte der 90er Jahre ein aktives Forschungsgebiet [Wol99, VS94, NB94, GCR99] der Constraintprogrammierung. Dynamische Constraintverarbeitung findet seine Anwendung z.B. in interaktiven Systemen, weil die interaktive Modellierung des Problems dort direkt im Constraintmodell umgesetzt werden kann. Durch die Dynamik des Solvers wird dann die bereits berechnete Lösung sofort an die veränderte Problemstellung angepasst. Aber auch bei der Implementierung der Constraintverarbeitung kann die Operation zur Constraintrücknahme sehr gut verwendet werden. Für die Integration von lokalen Suchverfahren und Propagation zum Beispiel sind dynamische Constraintlöser sehr geeignet [FLL01, Wol01].

Es gibt bisher aber kein System, mit dem man verteilte *und* dynamische Constraintverarbeitung inkrementell durchführen kann. Inkrementalität bedeutet, daß neue Eingaben jederzeit mit dem vorhandenen Wissen verarbeitet werden können, ohne dafür alle Berechnungen erneut ausführen zu müssen [FA97]. Dies ist eine wichtige Eigenschaft für viele Anwendungen, weil dadurch Rechenaufwand eingespart und die Benutzerinteraktion durch Ausgabe von Teilergebnissen verbessert werden kann. Insbesondere in kontinuierlichen Anwendungen, wie z.B. einem *truth-maintenance-system* [Doy79], sollen nur die notwendigen und nicht alle bisherigen Berechnungen beim Hinzufügen neuen Wissens durchgeführt werden. Alle derzeit existierenden dynamischen und inkrementellen Systeme können nicht verteilt verwendet werden, weil sie zentrale Datenstrukturen zur Constraintverarbeitung verwenden (z.B. [Wol99, GCR99]). Die Systeme zur nebenläufigen Constraintverarbeitung sind dagegen nicht inkrementell (z.B. [YD98, SSHF00]) oder nicht vollständig (z.B. [Han01]).

Ein Ausführungsmodell zur nebenläufigen und dynamischen Constraintverarbeitung würde die möglichen Anwendungsgebiete der Constraintprogrammierung erweitern. Damit könnten zum Beispiel verteilte interaktive Systeme, wie die verteilte Terminplanung, die im Folgenden beschrieben wird, deklarativ (weil constraintbasiert) implementiert werden. Ein solches System muß wegen der Nebenläufigkeit asynchrone Eingaben verarbeiten können und ist somit universeller einsetzbar als die meisten herkömmlichen Constraintlöser, die als zentrale Einheit alle Ereignisse synchronisieren und verarbeiten. Durch die Asynchronität sind die einschlägigen inkrementellen Ausführungsmodelle nicht anwendbar, weil sie einen synchronen Programmablauf voraussetzen, um z.B. ehemalige Zustände verwalten zu können. Ein globaler Zustand kann in nebenläufigen asynchronen Systemen oft gar nicht oder nur schlecht und fehleranfällig in Abhängigkeit von der Zeit (z.B. mit *time-stamps*) konstruiert werden.

Da zum Lösen der meisten für die Constraintprogrammierung relevanten Probleme nichtdeterministische Suchverfahren notwendig sind (weil die Probleme NP-vollständig sind), muß die Möglichkeit bestehen, falsche Entscheidungen rückgängig zu machen. Wenn ehemalige Zustände nicht zur Verfügung stehen, kann hierfür kein zustandsbasiertes *backtracking* verwendet werden, das bei Fehlern ehemalige Zustände wiederherstellt und dort alternative Entscheidungen trifft. Besteht aber die Möglichkeit der dynamischen Constraintverarbeitung, so können falsche Variableninstantiierungen im Nachhinein verändert werden, indem die falsche Instantiierung zurückgenommen und eine andere abgesetzt wird. Wenn das Absetzen von Constraints (und insbesondere von Instantiierungen) die einzige Möglichkeit ist, Variablendomänen zu verändern, dann kann mit seiner Umkehroperation, dem Zurücknehmen von Constraints, jede Fehlentscheidung rückgängig gemacht werden. Durch Constraintrücknahme erhält man zwar trotzdem keine ehemaligen Zustände[FFS98], kann aber *backtracking* mit den beiden genannten Zustandsüberführungsoperationen anwenden.

Weil Suchalgorithmen nicht auf globalen Systemzuständen, sondern auf einer Menge von abgesetzten oder zurückgenommenen Constraints operieren, ergeben sich viele Freiheiten bei der Wahl des angewandten Algorithmus [FFS98]. Die Suchalgorithmen können Instantiierungen, die Variablen an Werte binden, in beliebiger Reihenfolge verändern und sind nicht an die chronologische Abarbeitung gebunden. Man ist also nicht wie beim chronologischen *backtracking* darauf festgelegt, immer die letzte getroffene Instantiierung zurückzunehmen und zu verändern, sondern kann für jede beliebige Variable neue Werte wählen. Dadurch lassen sich, im Unterschied zu Erweiterungen von Prolog, auch lokale Suchverfahren [Voß00, FLL01] für dynamische Solver implementieren. Diese sehr effizienten (aber unvollständigen) Verfahren versuchen globale Lösungen zu finden, indem sie wiederholt lokale Veränderungen an initial zufällig gewählten Variablenbelegungen durchführen.

1.2 Angestrebte Ergebnisse

Zusammenfassung. Ein formales Ausführungsmodell für einen dynamischen Solver, der die verteilte Anwendung erlaubt soll definiert und verifiziert werden. Die speziellen Möglichkeiten des Modells zur dynamischen und verteilten Constraintverarbeitung und ein typisches Anwendungsfeld werden als *proof-of-concept* implementiert. Desweiteren wird eine operationale Semantik von Constraintprogrammen in diesem Ausführungsmodell abgeleitet und bzgl. der denotationalen Semantik verifiziert.

In dieser Arbeit soll ein Ausführungsmodell zur inkrementellen, dynamischen und nebenläufigen Constraintverarbeitung entwickelt werden. Darauf aufbauend soll ein constraintverarbeitendes System implementiert werden, das asynchrones Hinzufügen und Zurücknehmen von Constraints in parallelen Prozessen unterstützt. Das Ausführungsmodell soll unabhängig von speziellen Constraints und damit unabhängig von einer bestimmten Constraintdomäne definiert werden.

Aus einem Constraintprogramm kann eine eindeutige Menge von gültigen Constraints abgeleitet werden. Diese Menge schließt neben den explizit abgesetzten Constraints auch die (nichtdeterministisch gewählten) Instantiierungen mit ein, die ggf. nach erfolgreicher Suche Variablen an Werte binden. Die Propagationseigenschaften dieser Constraints können durch sogenannte Domänenreduktionsfunktionen charakterisiert werden. Durch die Chaotische Iteration [CM69] auf dieser Funktionsmenge in den Variablendomänen wird nach unendlich vielen Iterationsschritten ein Fixpunkt erreicht. Dieser Fixpunkt wird als denotationale Semantik eines Constraintprogramms betrachtet [Apt98, Rin02c].

Es soll gezeigt werden, daß das vorgestellte Ausführungsmodell ACS (*Asynchronous Constraint Solving*) diese denotationale Semantik berechnet. Dazu kann man aber nicht die Chaotische Iteration selbst zur Ausführung der in den Constraints implementierten Domänenreduktionsfunktionen verwenden, weil

1. zur Iteration manche Funktionen potentiell unendlich oft ausgeführt werden müssen,
2. keine Domänenerweiterung während der Iteration erlaubt ist,
3. in den Domänenreduktionsfunktionen keine Constraints abgesetzt werden dürfen.

Damit ACS praktisch anwendbar ist, sollte das Ausführungsmodell aber

1. in jedem Fall, in dem eine denotationale Semantik existiert, terminieren,
2. Domänenerweiterungen in die Iteration integrieren, damit die Rücknahme von Constraints möglich ist,
3. verschachteltes Absetzen von Constraints unterstützen, um praktisch relevante, komplexe Constraints zu ermöglichen.

Ein Ziel dieser Arbeit ist es also, ACS so zu konzipieren, daß diese Anforderungen erfüllt werden. Es muß daher gezeigt werden, daß die mit ACS berechnete operationale Semantik eines Constraintprogramms mit der denotationalen Semantik der darin geltenden Constraints übereinstimmt und unter den gleichen Bedingungen existiert.

Das Ausführungsmodell soll weiterhin so angelegt sein, daß auch verteilte Constraintverarbeitung möglich ist. Die definierten Algorithmen dürfen daher keine zentralen Datenstrukturen zur Inferenz des Ergebnisses verwenden und Kooperation zwischen parallel arbeitenden Constraintlösern ist nötig. In unterschiedlichen Agenten sollen Constraints beliebig abgesetzt oder zurückgenommen werden können, die kooperativ die Domänen gemeinsamer Variablen definieren.

Die effiziente Kombination von Propagation und Suche soll in ACS umsetzbar sein, damit die klassischen Anwendungsfelder der Constraintprogrammierung erreicht werden. Während der Ausführung eines Suchalgorithmus sollte daher Propagation durchgeführt werden können. Alternativ sollten aber auch andere Suchalgorithmen, wie z.B. lokale Suche [Voß00] oder Suche in einem gekapselten Suchraum [Sch00], integrierbar sein, um auch diese bei Bedarf anwenden zu können. In ACS sollen Suchalgorithmen daher in austauschbaren gekapselten Komponenten implementiert werden. So lassen sich vollständige und unvollständige Algorithmen verwenden, um eine ACS Implementierung dementsprechend als vollständigen bzw. unvollständigen Constraintlöser zu verwenden.

Die Objekt-orientierte Implementierung von ACS soll so aufgebaut sein, daß sie leicht um neue Constraintdomänen, Constraints und Suchalgorithmen erweiterbar ist. Mit dieser prototypischen Implementierung soll ein constraintbasiertes, interaktives und verteiltes Anwendungsbeispiel implementiert werden.

1.3 Fallstudie: Verteilte Terminplanung

Zusammenfassung. In diesem Abschnitt wird die Fallstudie motiviert und eingeführt, die im weiteren Verlauf der Arbeit als durchgängiges Beispiel beschrieben wird. Das Problem der Planung von Terminen, das im alltäglichen Leben fast jeder Person immer wieder zu lösen ist, wird als Constraintproblem modelliert und vorhandene Lösungen betrachtet. Die prototypische Implementierung $\mathcal{M}$Plan, ist ein neuer Ansatz für ein constraintbasiertes System von Software-Agenten zur Lösung dieses Problems der verteilten Terminplanung. Das Werkzeug $\mathcal{M}$Plan benutzt dazu die Techniken, die sich durch das asynchrone Ausführungsmodell zum Lösen dynamischer, kontinuierlicher und verteilter Constraintprobleme ergeben, welches Gegenstand dieser Arbeit ist.

1.3.1 Motivation

Die Planung von Terminen ist ein Problem, das jede Person im alltäglichen und vor allem im Geschäftsleben immer wieder zu lösen hat. Vielfältige Randbedingungen müssen oder sollen eingehalten werden, um alle gewünschten Termine wahrnehmen zu können. Jede Person plant seine Termine so, daß möglichst all seine persönlichen Randbedingungen (Constraints) eingehalten werden können.

Typische Constraints bei der Terminplanung einzelner Personen können zum Beispiel sein:

1. Die Zeiten unterschiedlicher Termine dürfen sich nicht überlappen.
2. Termine dürfen nur an dem Ort stattfinden, an dem man sich zu dieser Zeit aufhält.
3. Um zwei aufeinanderfolgende Termine wahrnehmen zu können, muß die notwendige Reisezeit dazwischen freigehalten werden.
4. Die Tagesarbeitszeit darf nicht überschritten werden, Pausen müssen eingehalten werden.
5. Es sollen so wenig wie möglich Ortswechsel zwischen den Terminen nötig sein.
6. Arbeitszeit soll für geschäftliche Termine, Freizeit für andere Termine verwendet werden.
7. Wichtige Termine sollen Vorrang haben.
8. Termine mit "wichtigen" Personen haben Vorrang.

Manche dieser Randbedingungen (die Punkte 1. bis 4.) sind unbedingt einzuhalten und werden als harte Constraints bezeichnet. Andere Constraints sollen möglichst erfüllt sein (die Punkte 5. bis 8.), sind aber nicht zwingend erforderlich. Solche Constraints, bei denen es qualitative Abstufungen der Erfüllung gibt, werden als weiche Constraints bezeichnet [BDFB+87, BMR97].

Die Schwierigkeit bei dieser Planung liegt darin, daß sich die Pläne unterschiedlicher Personen an bestimmten Punkten decken müssen. Wenn sich mehrere Personen zu einem Treffen verabreden, so muß dieses so in die Terminkalender aller Beteiligten eingetragen werden, daß die Randbedingungen aller Teilnehmerinnen und Teilnehmer eingehalten werden. Neben den persönlichen Randbedingungen ergeben sich also auch harte Constraints aus gemeinsamen Terminen:

9. Ich muß zur gleichen Zeit wie bestimmte andere Personen Zeit haben.
10. Ich muß zu einem Treffen mit anderen Personen am gleiche Ort sein wie sie.

Die Planung von gemeinsamen Terminen ist in der Praxis deshalb oft sehr schwierig, weil die Abstimmung der persönlichen Constraints mehrere Beteiligter sehr viel Kommunikation erfordert. Die Anzahl der erforderlichen Terminabsprachen zwischen zwei Personen wächst exponentiell mit der Anzahl der Personen, die an einem Treffen teilnehmen sollen. Es ist außerdem nicht wünschenswert, Personen, mit denen man sich verabreden möchte, alle persönlichen Randbedingungen mitzuteilen, damit ein gemeinsamer Termin gefunden werden kann. Man möchte normalerweise nur die nötigsten Informationen über die persönlichen Constraints in die Absprache einbringen. Außerdem wäre die Offenlegung aller persönlichen Constraints bei einem Terminplanungsproblem gar nicht möglich. Das Problem ist nämlich im Gegensatz zu den meisten bekannten Constraintproblemen nicht geschlossen darstellbar, sondern verändert sich für jede Person im Laufe der Zeit kontinuierlich. Es kommen ständig neue Constraints hinzu, während andere nicht mehr betrachtet werden müssen. Wenn man ein Treffen plant oder sich persönliche Randbedingungen ändern, ergeben sich neue Constraints. Constraints aus veralteten Randbedingungen und Constraints über vergangene Termine sollten dagegen gelöscht werden, um Kapazitäten freizugeben und die Planung nicht mehr zu beeinflussen.

1.3.2 Stand der Technik

Ein Software Werkzeug, das als Verbesserung des klassischen Terminkalenders aus Papier verwendet werden kann, sollte möglichst viel Kommunikation und Planung automatisch durchführen, ohne zu viel persönliche Informationen preiszugeben [KDK85, FMW01]. Aufgrund der alltäglichen Präsenz des Problems gibt es sehr viele Ansätze, mehr oder weniger "intelligente" Programme dafür zu entwickeln. Zur Wahrung der *privacy*, um also möglichst wenig persönliche Information herauszugeben, haben sich Agenten-Architekturen für solche Werkzeuge durchgesetzt [GS96]. Dies sind offene Systeme, die aus mehreren Agenten bestehen, die jeweils einen Terminkalender verwalten. Neue Agenten können problemlos in eine solche Architektur eingebunden werden und alle persönlichen Informationen werden lokal verwaltet.

Von der Firma Microsoft wird ein digitaler Terminkalender mit dem eMail Werkzeug MS Outlook [MSO] vertrieben. Die persönlichen Präferenzen des Benutzers werden hier nicht explizit eingegeben, sondern müssen bei der Festlegung von Terminen jeweils durch den Benutzer berücksichtigt werden. In MS Outlook findet also keine automatische Planung statt. Das Werkzeug sichert allerdings, daß die harten Constraints über mehrere Teilnehmern (Punkte 9 und 10 auf Seite 7) eingehalten werden. Dies wird bewerkstelligt, indem jeder Benutzer bevor er ein Treffen mit einer anderen Person festlegt, in dessen Terminkalender prüfen kann, ob noch ein freier Zeitraum für das Treffen besteht. Der Nachteil dieser interaktiven Planung besteht darin, daß man damit seinen Terminpartnern preisgibt, wie viel Zeit man an dem bestimmten Tag hat, wobei die Information, ob man Zeit hat, eigentlich ausreichen würde. Eine ähnliche Funktionalität weist auch der Meetingmaker [Meea] auf, der als kommerzielles Werkzeug erhältlich ist. Die Planung erfolgt auch hier durch den Anwender, der die Möglichkeit hat, freie Zeiten anderer Benutzer für ein Treffen zu reservieren.

In anderen Werkzeugen, wie z.B. dem Meeting Wizard [Meeb], wird auf die Planung vollständig verzichtet und statt dessen ein Rahmen gegeben, in dem Vorschläge gemacht und beantwortet werden können. Das System beruht auf dem Verschicken von eMails, die die nötige Information zum Treffen von Verabredungen enthalten. Die Planung der Vorschläge muß dann jeder Benutzer selbst durchführen. Bei diesen eMail-basierten Systemen findet keinerlei automatische Unterstützung der Planung statt. Weil man hier nicht vorher nachsehen kann, ob ein Treffen für jemanden anders überhaupt möglich ist, ist die *privacy* für jeden Benutzer voll gesichert.

Die akademischen Arbeiten zu diesem Gebiet, z.B. [GS96, SD98, FMW01, SSHF01], versuchen unter der Einhaltung der *privacy* möglichst viel automatische Planung durchzuführen. Es sind allerdings zur Zeit aus diesem Bereich keine Implementierungen erhältlich. Die Idee, Constraint-basierte Verfahren zu verwenden, scheint sich aber durchgesetzt zu haben, spezialisierte Ansätze aus der verteilten Künstlichen Intelligenz (z.B. [EZR94]) werden kaum noch verfolgt.

1.3.3 Die Implementierung $\mathcal{M}$Plan

Mit dem System $\mathcal{M}$Plan habe ich eine prototypische Implementierung für ein verteiltes, Agenten-basiertes Software-Werkzeug zur Verwaltung und Planung von Terminen entwickelt. Die Anwendung beruht auf der Idee, Termine so zu verwalten, daß Constraints, wie sie auf Seite 7 dargestellt sind, immer eingehalten werden. Dazu werden Termine als Objekte modelliert, deren Eigenschaften durch Constraints beschrieben werden. Solche Eigenschaften sind z.B. Anfangszeitpunkt, Dauer oder Ort des Treffens. Jedes Termin-Objekt repräsentiert eindeutig ein Treffen, das von einem Benutzer vorgeschlagen wurde. Über dieses Objekt wird dann die nötige Kommunikation weitgehend automatisch durchgeführt. Für die Einhaltung der Constraints ist ein Constraintlöser zuständig, der in jedem Agent angelegt wird, wie es in Abb. 1.1 dargestellt ist. Der Constraintlöser definiert die Eigenschaften der Treffen so, daß immer alle harten und möglichst viele der weichen Constraints erfüllt sind.

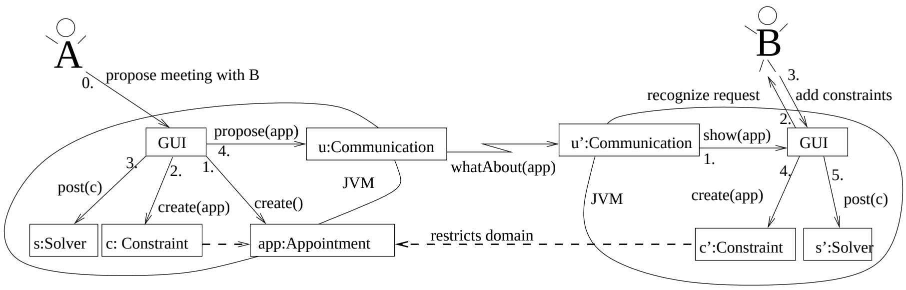

Abbildung 1.1: Ein typisches Szenario für verteilte Terminplanung. Der Benutzer A eines Agenten schlägt einen Termin app mit dem Benutzer B eines anderen Agenten vor. Die Randbedingungen c und c' beider Teilnehmer werden durch die Constraintlöser s bzw. s' berücksichtigt.

Die persönlichen Präferenzen und Kalender brauchen in dieser Architektur nicht offengelegt zu werden, weil diese im lokalen Constraintlöser gehalten werden. Andere Benutzer des Systems können lediglich Information über die Treffen erhalten, an denen sie selbst teilnehmen. Die via Internet erreichbaren Termin-Objekte können von berechtigten Benutzern beliebig gelesen werden, so daß im Unterschied zu allen bekannten Systemen die Verwaltung eines gemeinsamen Kalenders in mehreren Agenten möglich ist. Dazu muß lediglich die relevante Information verschlüsselt an die unterschiedlichen Agenten einer Person, die z.B. auf seinem Computer, dem seiner Sekretärin und seinem Notebook laufen, weitergeleitet werden.

Eine genauere Beschreibung sowohl der Modellierung des Terminplanungssystems, also auch seiner Lösung mit $\mathcal{M}$Plan werde ich im Verlauf der Arbeit als Fortsetzung der Fallstudie geben. Der verwendete Constraintlöser ist J.CP, der das Ausführungsmodell des asynchronen Constraintlösens implementiert. Ich werde bei der Definition und Analyse von ACS an einigen Stellen die Relevanz und Anwendung der theoretischen Aspekte unter anderem am Beispiel der verteilten Terminplanung angeben.

1.4 Gliederung der Arbeit

Im nächsten Kapitel werden einige Grundlagen der Constraintverarbeitung zusammengefaßt. Die Problemklasse, mit der sich die Arbeit beschäftigt und einige bekannte Konzepte und deren Implementierungen zum Lösen solcher Probleme werden dargestellt. Schließlich gebe ich noch eine kurze Einführung in die Theorie der Chaotischen Iteration, die wesentlich für die theoretische Analyse des Beitrags dieser Arbeit ist.

In Kapitel 3 wird das Ausführungsmodell ACS zur asynchronen Constraintverarbeitung, das Gegenstand dieser Arbeit ist, definiert und verifiziert. Es wird dargestellt, wie ACS durch Constraintprogramme benutzt wird, in denen Constraints abgesetzt oder zurückgenommen werden können. Die Bedingungen an die Eingabe zur Terminierung der vorgestellten Algorithmen werden angegeben sowie die Vollständigkeit und Korrektheit bzgl. des Ergebnisses der Chaotischen Iteration nachgewiesen.

In Kapitel 4 wird die prototypische Implementierung des Ausführungsmodells beschrieben. Besonderes Augenmerk wird hier auf die Erweiterbarkeit um neue Constraints und Constraintsysteme gelegt.

Das fünfte Kapitel beschreibt, wie Suche in ACS und seine Implementierung integriert wird. Es wird angegeben, wie Suchalgorithmen als Constraints in das System integriert werden können und welche Eigenschaften sie haben müssen, um Terminierung und Vollständigkeit zu gewährleisten. Die beiden implementierten Suchalgorithmen werden detailliert dargestellt.

In Kapitel 6 wird die Eignung des Ausführungsmodells für verteilte Anwendungen untersucht. Die besonderen Probleme, die sich aus der parallelen Verarbeitung ergeben, werden dargestellt und es wird aufgezeigt, wie sie in der prototypischen Implementierung gelöst werden.

In Kapitel 7 wird die Laufzeit des prototypischen Constraintlösers untersucht. Es werden interne Vergleiche zwischen unterschiedlichen Teilkomponenten und Vergleiche mit anderen erhältlichen Systemen angestellt. Auch die Funktionalität der anderen Systeme wird mit der des vorgestellten Systems verglichen.

Im letzten Kapitel fasse ich die wesentlichen Ergebnisse der Arbeit zusammen und gebe einen Ausblick auf Anwendungsbereiche und Erweiterungsmöglichkeiten.

Kapitel 2

Grundlagen

In diesem Kapitel wird der Stand der Wissenschaft auf dem Gebiet des Constraintlösens dargestellt. Aus der Literatur werden die für ACS relevanten Definitionen zitiert und teilweise verallgemeinert, um in der Beschreibung von ACS im nächsten Kapitel verwendet werden zu können. In Abschnitt 2.1 werden grundsätzliche Begriffe zur Definition von *Constraint Satisfaction* Problemen angegeben. Danach werden unterschiedliche Konzepte wie Suche und Propagation zum Lösen dieser NP-vollständigen Probleme vorgestellt und ihre Verwendung in Constraintverarbeitenden Systemen erläutert. In Abschnitt 2.4 wird die Theorie der Chaotischen Iteration vorgestellt, die die Grundlage der denotationalen Semantik für ACS Constraintprogramme bildet.

Constraintprogrammierung (CP) basiert auf der Verarbeitung von Constraints und dem Lösen von Constraintproblemen. Constraints beschreiben die Eigenschaften von Variablen und werden in constraintbasierten Modellen bzw. Constraintproblemen konjunktiv verknüpft. Dabei können mehrere Constraints die Eigenschaften gemeinsamer Variablen beschreiben und somit nicht triviale Probleme dadurch definieren, daß alle Constraints erfüllt sein sollen. Constraintprobleme werden durch Constraintprogramme implementiert, indem die Constraints in einer Sprache formuliert werden, die ein (z.B. in eine Programmiersprache integrierter) Constraintlöser verarbeiten kann. Der Constraintlöser verarbeitet die Implementierung analog zu einem Ausführungsmodell, das die Semantik der Sprache operational umsetzt. Die häufigsten Ziele sind dabei:

1. die Erfüllbarkeit (*satisfaction*) von Constraintproblemen festzustellen. Dabei ist zu prüfen, ob alle Variablen so instantiiert werden können, daß alle Constraints erfüllt sind

2. die Vereinfachung (*simplification*) von Constraintproblemen. Constraints werden so verarbeitet, daß Lösungen leichter gefunden werden können, indem z.B. der Suchraum des Problems eingeschränkt wird

3. die Optimierung (*optimization*) von Lösungen eines Constraintproblems. Es wird ein Kriterium angegeben, nach dem sich die Qualität von Lösungen vergleichen läßt und eine diesbezüglich optimale Lösung gesucht

Diese Arbeit beschäftigt sich dabei hauptsächlich mit dem zweiten Punkt, also der Vereinfachung von Constraintproblemen, wobei Punkt 1, also die Erfüllbarkeit, als Spezialfall betrachtet wird: Wenn sich ein Problem P, erweitert um Instantiierungen für jede Variable, vereinfachen läßt, ohne daß dabei Inkonsitenzen auftreten, dann ist P erfüllbar.

2.1 Constraintprobleme

Zusammenfassung. Grundlegende Begriffe der Constraintprogrammierung wie Constraintvariable und Constraint werden definiert. Jeder Variablen wird eine Menge zugeordnet, die alle Werte enthält, die sie annehmen kann. Constraints beschreiben diese Mengen von Werten, so daß Constraintsequenzen die zulässigen Werte gemeinsamer Variablen konjunktiv beschreiben. Durch diese Interpretation kann die Erfüllbarkeit von Constraintsequenzen für jede Variablenbelegung entschieden werden.

Als Basis der Constraintverarbeitung wird meistens ein Universum angegeben, in der die Constraints eine deklarative Semantik haben. Dieses wird i.d.R. durch eine Menge oder eine Algebra definiert, in der Constraints wie eine mathematische Relation die Beziehung von Variablen beschreiben. Die meisten Constraintlöser stellen Constraints, die in einem solchen Universum eindeutig formulierbar sind, zur Implementierung von Constraintproblemen zur Verfügung. Bekannte Beispiele solcher Domänen aus der Constraintverarbeitung sind z.B. die der endlichen Mengen ganzer Zahlen (z.B. [CD96]), die der Intervalle reeller Zahlen (z.B. [JMSY92]) oder endliche Mengen beliebiger unterscheidbarer Objekte (z.B. [JCL]). Es ist allerdings nicht unbedingt notwendig, jedes Constraintproblem in einer solchen gegeben Domäne zu modellieren, wenn man das Verhalten von Constraints problembezogen definieren kann und nicht die abstrakten Bibliotheken anderer Systeme verwendet. Ein regelbasierter Ansatz ohne die Verwendung von expliziten Domänen ist z.B. die Sprache CHR [Frü98], in der die möglichen Werte der Variablen durch Constraints beschrieben werden. In meinem Ansatz verwende ich Variablen als eigenständige aktive Objekte, die die Menge ihrer zulässigen Instantiierungen explizit verwalten. Daher definiere ich explizite Domänen, in denen Constraints deklarativ als Relationen interpretiert werden können, durch (beliebige) Mengen die sich eindeutig beschreiben lassen.

Definition 2.1.1 (Constraintdomäne) Eine *Constraintdomäne* D ist eine Menge, so daß für jede Teilmenge $D' \in \mathcal{P}ow(D)$ eine Mengenbeschreibung $\overline{p} \in P_1 \times P_2 \times ...P_n$ existiert. Dabei sind P_i Merkmale der Mengen $D' \in \mathcal{P}ow(D)$, $\overline{p}$ ein Merkmalsvektor, und $\mathsf{dom}(\overline{p}) = D'$ eine surjektive Abbildung. D' heißt dann *Interpretation* von $\overline{p}$ und $\overline{p}$ heißt *Domänenbeschreibung* von D'.

Domänenbeschreibungen sind in der Regel nicht eindeutig. In Implementierungen von Constraintverarbeitenden Systemen wird aus Effizienzgründen oft zwischen unterschiedlichen Beschreibungen gleicher Mengen gewechselt. Ich gehe davon aus, daß sich jede Domäne beschreiben läßt und daß jeder Beschreibung eindeutig eine Menge zugeordnet werden kann.

Die Beschreibung der aktuell zulässigen Werte einer Variablen wird in den meisten constraintverarbeitenden Systemen (z.B. GNUProlog [Gnu] oder ILOG [ILOG]) explizit in jeder Constraintvariable gespeichert. Die Interpretation dieser Wertemenge enthält immer die Werte, die nach aktuellem Wissen möglicherweise noch zu einer Lösung des aktuellen Constraintproblems führen können. Constraints begrenzen die Menge dieser Werte, um das Finden einer Lösung zu vereinfachen bzw. um die Erfüllbarkeit des Problems nachzuprüfen (Punkte 2 und 1 auf Seite 11).

Aus technischen Gründen verwalten Constraintvariablen neben ihrer aktuellen Domäne auch Referenzen auf Constraints, die die Eigenschaften ihrer Domänen mit denen anderer Variablen in Beziehung setzen. Auf die Verwendung dieser Referenzen beim asynchronen Constraintlösen werde ich in Kapitel 3 eingehen.

Definition 2.1.2 (Constraintvariable) Eine *Constraintvariable* $v = (\overline{p}, C)$ einer Constraintdomäne D ist gegeben durch

- eine Domänenbeschreibung $\overline{p} \in P_1 \times P_2 \times ... \times P_n$ mit $\mathsf{dom}(\overline{p}) \subseteq D$
- eine Merkmal-indizierte Familie C von Sequenzen C_P von Constraints: $C = (C_P)_{P \in \{P_1,...,P_n\}}$.

Die *Domäne* von v ist die Menge $\mathsf{dom}(\overline{p})$, man schreibt auch $\mathsf{dom}(v) = \mathsf{dom}(\overline{p})$

Bemerkung 2.1.1
Die Constraints aus jeder Sequenz C_P sind diejenigen Constraints über v, die das Merkmal P verändern können. Dies ist für die Implementierung relevant, vom deklarativen Standpunkt kann man hier auch leere Sequenzen annehmen.

Die Werte, die Constraintvariablen annehmen können, ohne dabei die Erfüllbarkeit eines Constraintproblems zu verhindern, werden durch die Constraints des Problems selbst beschrieben. Jedes Constraint bezieht sich auf eine geordnete Menge von Variablen, deren zulässige Belegungen es beschreibt. Ein Constraint wird also formal als mathematische Relation, also als Teilmenge des Kreuzproduktes der Constraintdomänen seiner Variablen, definiert. Diese Definition von Constraints ist recht allgemein und rein deklarativ, denn zur Implementierung ist eine solche explizite Handhabung von Vektoren zu ineffizient. Dennoch zitiere ich diese Definition aus z.B. [Apt98], weil sie jedes beliebige Constraint beschreibt.

Definition 2.1.3 (Constraint) Gegeben seien die Variablen $v_1, v_2, ..., v_n$ aus den Constraintdomänen $D_1, D_2, ..., D_n$, so daß $\mathsf{dom}(v_i) \subseteq D_i$ gilt. Ein *Constraint* c über dem Variablenvektor $(v_1, ..., v_n)$ definiert eine Menge von Vektoren $D \subseteq D_1 \times D_2 \times ...D_n$. Dabei heißen $\mathsf{vars}(c) := (v_1, ..., v_n)$ die Variablen und $\mathsf{sem}(c) := D$ die deklarative Semantik von c.

Bemerkung 2.1.2
In den meisten Constraints und Constraintverarbeitenden Systemen sind die Constraintdomänen aller verwendeten Variablen gleich ($D_1 = D_2 = ... = D_n$). Zugunsten der Allgemeinheit verzichte ich auf diese Beschränkung und erlaube auch Constraints über Variablen verschiedener Constraintdomänen, sogenannte heterogene Constraints [BS98].

Fallstudie, Teil 1

Bei der verteilten Terminplanung $\mathcal{M}$Plan werden die Termine durch mehrere Constraintvariablen modelliert, die die Eigenschaften des Termins beschreiben. Das sind:

- *Tag des Termins, ein Wert aus der unendlichen Menge aller Datumsangaben, die sich durch die Eigenschaften Tag, Monat und Jahr beschreiben lassen:* $tag \in \{1..31\} \times \{1..12\} \times \mathbb{Z}$
- *Beginn des Termins, ein Wert aus der Constraintdomäne der Uhrzeiten, bestehend aus Stunde und Minute:* $zeit \in \{0..24\} \times \{0..59\}$
- *Dauer des Termins, ein Wert der in Minuten angegeben wird:* $dauer \in \mathbb{N}$
- *Ort, an dem der Termin stattfindet, eine Zeichenkette, die von den Benutzern interpretiert werden können sollte:* $ort \in String$

Constraints über Termine beziehen sich dann auf die Variablen, die die Termine beschreiben. Das Constraint, das sichert, daß sich zwei Termine nicht überschneiden, kann dann folgendermaßen als Relation dargestellt werden:
$disjoint((tag, zeit, dauer, ort), (tag', zeit', dauer', ort'))$
$:= \{(tag, zeit, dauer, ort, tag', zeit', dauer', ort') \mid tag = tag' \Rightarrow (zeit + dauer < zeit' \vee zeit' + dauer' < zeit)\}$

Durch eine Modellierung von Constraints und Constraintvariablen kann jedes Constraintproblem durch ein sogenanntes Constraintnetzwerk [Mon74] dargestellt werden.

In einem Graphen mit Hyperkanten werden Variablen als Knoten und Constraints als Kanten dargestellt. Diese Modellierung hat sich wegen ihrer Anschaulichkeit zur Dokumentation von Constraintproblemen etabliert. Bevorzugt wird diese Darstellung gewählt, um ausschließlich binäre Constraints, also Constraints über genau 2 Variablen, einer gemeinsamen Constraintdomäne zu beschreiben. Es ist aber durch die Verwendung von Hyperkanten auch die Darstellung globaler Constraints (über mehr als 2 Variablen) möglich. In heterogenen Constraintnetzen, also Modellen von Constraintproblemen mit Variablen verschiedener Constraintdomänen, sind die unterschiedlichen Domänen dabei implizit durch die verwendeten Constraints gegeben. In Fallstudie Teil 2 gebe ich beispielhaft ein Constraintnetz an, das ein Teilproblem einer fiktiven Anwendung der Terminplanung aus Abschnitt 1.3 modelliert.

Fallstudie, Teil 2

Abb.2.1 zeigt einen Teil des Constraintnetzes eines konkreten Terminplanungsproblems. Die Termine sollen sich paarweise nicht überschneiden, Termin 1 soll vor Termin 2 stattfinden und die Termine 2 und 3 sollen in der Benutzerschnittstelle graphisch angezeigt werden. Die Constraintdomäne der Benutzeroberfläche GUI kann als die Menge der anzuzeigenden Termine definiert werden. Letztere Verwendung von Constraints ist in der CP nicht üblich, weil hierdurch keine Domänenbeschränkungen definiert werden. Mit den gegebenen Definitionen kann man aber auch solche Beziehungen modellieren. In $\mathcal{M}$Plan werden Constraints tatsächlich verwendet, die die Möglichkeit nutzen bestimmte Methoden erneut auszuführen, wenn eine Variable instantiiert wurde. In diesem Beispiel wird das Constraint "zeigt an" und damit die Methode zur Anpassung der GUI immer dann erneut ausgeführt, wenn ein Termin festgelegt wurde.

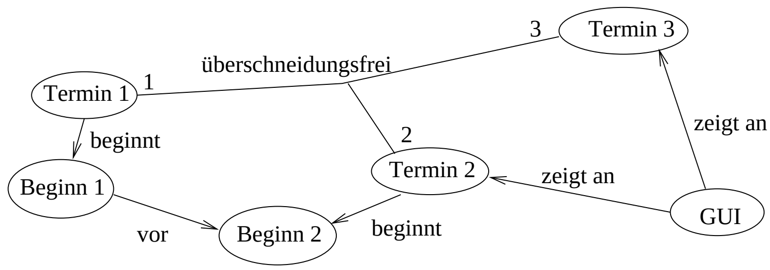

Abbildung 2.1: Constraintnetz aus der Terminplanung

Um den Effekt von Constraints innerhalb eines Constraintproblems vollständig beschreiben zu können, definiert man oft den globalen Zustandsraum eines Constraintproblems. In der Constraintlogischen Programmierung z.B. werden alle logischen Variablen zentral in Beziehung zur Abarbeitung eines Programms verwaltet, so daß sich die Auswirkungen der Bearbeitung von Zielen an den veränderten Domänen der Variablen abgelesen werden können (z.B. [CD96]). Ich definiere den globalen Zustandsraum eines Constraintproblems (wie z.B. [Apt98]), ohne ihn später operational umzusetzen. Durch den Begriff des Constraintschemas in der folgenden Definition wird lediglich ein theoretischer, globaler Raum geschaffen, in dem sich die Auswirkungen der Constraints beschreiben lassen. Dazu definiere ich zunächst eine Abbildung von Vektoren auf Mengen, die alle Elemente des Vektors enthalten.

Definition 2.1.4 (Vektoren, Sequenzen, Mengen) Die Abbildung ε, ordnet jedem Vektor und jeder Sequenz eindeutig eine Menge zu: Sei $(x_1, ..., x_n)$ eine Vektor, dann ist $\varepsilon((x_1, ..., x_n)) := \{x_1, ..., x_n\}$. Sei $\langle x_1, ..., x_n \rangle$ eine Sequenz, dann ist $\varepsilon(\langle x_1, ..., x_n \rangle) := \{x_1, ..., x_n\}$.

Durch diese Abbildung können die Variablen der Constraints auf Mengen projeziert werden, in denen durch Vereinigung doppelte Variablen eliminiert werden. Durch eine Ordnung der verbleibenden Variablen ergibt sich ein Vektor, der jede Variable, jedes Constraints eines Constraintsproblems genau einmal enthält.

Definition 2.1.5 (Constraintschema) Gegeben sei eine Menge von Constraints C. Ein *Constraintschema* von C ist eine Sequenz von Variablen $\langle v_1, ..., v_m \rangle$, so daß $\bigcup_{c \in C} \varepsilon(\mathsf{vars}(c)) \subseteq \varepsilon(\langle v_1, ..., v_n \rangle)$ und $\forall i, j \in 1..n : v_i \neq v_j$ gilt.

Aus einer geordneten Menge, also einer Sequenz, von Constraints kann man solche Constraintschemata eindeutig ableiten, weil die Variablen eines Constraints ebenfalls geordnet sind. Dies kann auf unterschiedliche Weise geschehen. Für die Verwendung von Constraintschemata zur eindeutigen Beschreibung von Systemzuständen ist lediglich darauf zu achten, daß die Konstruktion des Schemas aus einer Constraintsequenz eindeutig ist. In Def. 2.1.6 definiere ich eine

Abbildung, die alle Variablen in einer Sequenz akkumuliert, indem alle Constraints der Reihe nach untersucht werden und nur solche Variablen im Ergebnis berücksichtigt werden, die nicht in späteren Constraints vorkommen.

Definition 2.1.6 (generiertes Constraintschema) Sei eine Sequenz von Constraints $C = \langle c_1, ..., c_n \rangle$ gegeben, dann ist das *generierte Constraintschema* von C definiert durch $\mathsf{schem}(C) := \mathsf{flatten}(\mathsf{vars}(c_1) \circ ... \circ \mathsf{vars}(c_n))$ wobei gilt

$$\mathsf{flatten}(\langle v_1, ..., v_n \rangle) = \begin{cases} \langle\rangle, \text{ falls } n = 0 \\ \langle v_1 \rangle \circ \mathsf{flatten}(\langle v_2, ..., v_n \rangle), \text{ falls } n > 0 \land v_1 \notin \varepsilon(\langle v_2, ..., v_n \rangle) \\ \mathsf{flatten}(\langle v_2, ..., v_n \rangle) \text{ ,sonst} \end{cases}$$

In solchen globalen Constraintschemata kann die Reihenfolge der Variablen genutzt werden, um Belegungen aller Variablen durch Vektoren zu beschreiben. Die Belegung selbst kann dann als punktweise Abbildung der Variablen in die Menge der zulässigen Werte interpretiert werden, wie es in der Literatur üblich ist [EM85, FA97]. Dabei sind die Mengen zulässiger Werte durch die Constraintdomänen der Variablen gegeben.

Definition 2.1.7 (Variablenbelegung) Sei ein Variablenvektor $(v_1, ..., v_n)$ aus den Constraintdomänen $D_1, ..., D_n$ mit $\mathsf{dom}(v_i) \subseteq D_i$ gegeben, dann ist jeder Vektor $(d_1, ..., d_n) \in D_1 \times ... \times D_n$ eine *Variablenbelegung (von $(v_1, ..., v_n)$).*

Für Variablenbelegungen des generierten Schemas eines Constraints kann die Erfüllbarkeit des Constraints, wie sie auf Seite 11 beschrieben wurde, entschieden werden.

Definition 2.1.8 (Erfüllen von Constraints) Eine Variablenbelegung $\overline{d} \in D_1 \times ... \times D_n$ *erfüllt* ein Constraint c mit $\mathsf{vars}(c) = (v_1, ..., v_n)$ und $\mathsf{dom}(v_i) \subseteq D_i$, gdw. $\overline{d} \in \mathsf{sem}(c)$ gilt.

Um die Erfüllbarkeit einer Menge bzw. einer Sequenz von Constraints entscheiden zu können, verwende ich den globalen Zustandsraum, der durch das generierte Schema konstruiert wurde. Es werden für jedes Constraint die relevanten Variablen aus dem globalen Schema herausprojeziert, um dann für jedes Constraint die Erfüllbarkeit zu entscheiden.

Definition 2.1.9 (Erfüllen von Constraintsequenzen) Sei eine Sequenz von Constraints C gegeben, so daß $\mathsf{schem}(C) = \langle v_1, ..., v_n \rangle$ gilt. Dann *erfüllt* eine Variablenbelegung $\overline{d}$ von $(v_1, ..., v_n)$ die Constraintsequenz C gdw. $\forall c \in C$: $\overline{d} \downarrow \mathsf{vars}(c) \in \mathsf{sem}(c)$.

Die Erfüllbarkeit von Constraintsequenzen wird also durch die Erfüllbarkeit aller vorkommenden Constraints definiert. Daraus ergibt sich, daß man diesen Erfüllbarkeitsbegriff für Constraintprobleme als Konjunktionen von Constraints verwenden kann. Solche *Constraint Satisfaction* Probleme [Hen89, MR98] (CSP) sind Gegenstand dieser Arbeit.

Definition 2.1.10 (Constraint Satisfaction Problem) Sei C eine Sequenz von Constraints mit dem generierten Schema $\mathsf{schem}(C) = \langle v_1, ..., v_n \rangle$ und $D = \langle D_1, ..., D_n \rangle$ eine Sequenz von Constraintdomänen, so daß $\mathsf{dom}(v_i) \subseteq D_i$ gilt, dann ist das Paar (D, C) ein *Constraint Satisfaction Problem.*

Bemerkung 2.1.3
Mit dem Begriff des generierten Constraintschemas und der Annahme, daß die Constraintdomäne jeder Variablen bekannt ist, kann man ein CSP auch als einfache Sequenz oder Menge von Constraints definieren. Aus Gründen der Konvention verzichte ich auf diese Vereinfachung.

Constraintsequenzen bzw. CSPs, die nicht erfüllbar sind, heißen inkonsistent (z.B. [MS98]). Wird ein Constraintproblem durch ein inkonsistentes CSP modelliert, so ist das Problem nicht lösbar (vorausgesetzt, daß die Modellierung korrekt und vollständig ist).

Definition 2.1.11 (inkonsistent) Eine Sequenz von Constraints C heißt *inkonsistent*, wenn es keine Variablenbelegung gibt, die C erfüllt.

Ist ein CSP nicht inkonsistent in diesem Sinne, so ist es erfüllbar, hat also eine Lösung. Jede Variablenbelegung, die alle Constraints erfüllt, ist eine Lösung.

Definition 2.1.12 (Lösung) Sei $P = (\langle D_1, ..., D_n\rangle, C)$ ein CSP, dann ist eine Variablenbelegung $\overline{d} \in D_1 \times ... \times D_n$ eine *Lösung* von P, wenn sie C erfüllt. Die Menge aller Lösungen von P heißt $\mathsf{sol}(C)$.

Fallstudie, Teil 3
Ein Terminplanungsproblem mit drei Terminen t_1, t_2 und t_3, die sich nicht überschneiden sollen, könnte z.B. durch folgende Constraints gegeben sein:

$$C = \langle disjoint(t_1, t_2), disjoint(t_2, t_3), disjoint(t_1, t_3)\rangle$$

Daraus wird das Constraintschema $\mathsf{schem}(C)$ *folgendermaßen generiert:*

$$\langle tag_2, zeit_2, dauer_2, ort_2, tag_1, zeit_1, dauer_1, ort_1, tag_3, zeit_3, dauer_3, ort_3\rangle$$

Angenommen, die Termine sollen alle am 20.12.2002 in Berlin stattfinden und es gäbe bereits die Einschränkungen, daß t_1 und t_2 nicht vormittags und t_3 nicht vor 14 Uhr beginnen sollen, dann ergibt sich folgendes CSP:

$$\begin{array}{l} P := (\langle\{(20, 12, 2002)\}, \{(h, m) \mid h > 12\}, \{1..720\}, \{Berlin\}, \\ \{(20, 12, 2002)\}, \{(h, m) \mid h > 12\}, \{1..720\}, \{Berlin\}, \\ \{(20, 12, 2002)\}, \{(h, m) \mid h > 14\}, \{1..600\}, \{Berlin\}\rangle, C) \end{array}$$

Ein Lösung von P wäre dann eine Instantiierung der Variablen, so daß die Termine folgendermaßen stattfinden: t_1 von 12 bis 12Uhr45, t_2 von 13 bis 14 Uhr und t_3 von 14Uhr15 bis 15 Uhr. Die Variablenbelegung, die diese Lösung repräsentiert wäre dann:

$$\begin{array}{l} \langle(20, 12, 2002), (13, 0), 60, Berlin, (20, 12, 2002), (12, 0), 45, Berlin, \\ (20, 12, 2002), (14, 15), 45, Berlin\rangle \end{array}$$

2.2 Constraintlösen und Constraintlöser

Zusammenfassung. *Constraint Satisfaction* Probleme sind im allgemeinen NP-vollständig, so daß nichtdeterministische Verfahren erforderlich sind, um Lösungen zu finden bzw. nachzuweisen, daß es keine Lösung gibt. In diesem Abschnitt werden bekannte grundlegende Verfahren zum Lösen von CSPs diskutiert. Paradigmen zur Suche wie *generate-and-test* oder lokale Suchverfahren und Propagationstechniken wie *forward checking* und *look ahead* werden vorgestellt und bzgl. ihrer Eignung für Constraintlöser diskutiert.

Die Existenz von Lösungen eines CSP kann im Allgemeinen nicht aus dem CSP direkt abgeleitet werden (z.B. [Wal75]). Da der Hauptanwendungsbereich der Constraintprogrammierung in der Bearbeitung von NP-vollständigen Problemen liegt, erfordert das Finden von Lösungen und der damit verbundene Existenzbeweis nichtdeterministische Verfahren, sogenannte Suchalgorithmen. In NP-vollständigen Problemen existiert (nach dem heutigen Wissensstand) kein deterministischer Algorithmus zum Finden von Lösungen [CS88]. Es lassen sich natürlich auch Aufgaben "einfacherer" Problemklassen durch Constraints modellieren und lösen, allerdings können dort i.d.R. Lösungen durch spezielle Algorithmen effizienter (*greedy*) berechnet werden. Trotzdem können auch hier Verfahren der Constraintverarbeitung effizient eingesetzt werden, wenn diese lokale Optimierungskriterien verwenden, wie es z.B. für die bekannte Gruppe der SAT (*satisfiability*) Probleme in [Wal00] untersucht wurde.

Zum Lösen allgemeiner CSPs mit endlichen Constraintdomänen werden nichtdeterministische Suchverfahren verwendet. Dabei werden Variablenbelegungen generiert und jeweils überprüft wird, ob diese in der deklarativen Semantik aller Constraints enthalten sind. Diese *generate-and-test*-Methode kann am einfachsten implementiert werden, indem man einen Konsistenztest für alle abgesetzten Constraints verwendet. Ein solcher Test überprüft, ob alle Constraints durch eine gegebene Variablenbelegung erfüllt sind. Durch die Verwaltung expliziter Relationen wie z.B. in JCL [JCL] oder im *Relational Constraint Solver (RCS)* [See01] kann die Erfüllbarkeit der Constraints geprüft werden. Alternativ kann eine implizite Beschreibung von Relationen bzw. Prädikaten, wie in Prolog [SS94], verwendet werden, um die Konsistenz zu prüfen. Solche Verfahren sind aber für die meisten Constraintprobleme zu ineffizient, weil der Suchraum, also die Menge der zu untersuchenden Variablenbelegungen, zu groß ist. Der Suchraum von CSPs wächst exponentiell mit der Anzahl der Variablen und ihren möglichen Werten, so daß die Erzeugung aller möglichen Variablenbelegungen schon bei relativ kleinen Problemen praktisch unmöglich wird.

Wenn eine Variablenbelegung eruegt wurde, die keine Lösung im Sinne von Def. 2.1.12 eines Constraintproblems ist, so kann man versuchen diese Variablenbelegung durch gezielte lokale Veränderungen so anzupassen, daß eine Lösung entsteht. Diese Verfahren werden in sogenannten *local search* Algorithmen implementiert [Voß00, PG96, Nar01]. Wenn eine zufällig erzeugte Variablenbelegung das gegebene CSP nicht erfüllt, so wird die nächste zu untersuchenden Variablenbelegung nicht wie bei *generate-and-test* abhängig von der Struktur der Variablendomänen ausgewählt, sondern es werden Veränderungen der aktuellen Belegung untersucht und weiterverfolgt. Dazu werden ähnliche Belegungen aus der aktuellen Belegung erzeugt und miteinander mittels einer bestimmten

Fehlerfunktion verglichen. Dann wird diejenige Belegung weiterentwickelt, die dabei den geringsten Fehler hatte. Die Auswahl der alternativen Variablenbelegung erfolgt bei *local search* also abhängig vom Fehler der aktuellen Belegung und nicht nach einer a priori festgelegten Reihenfolge. Dadurch kann man die Menge aller Variablenbelegungen nur schwer vollständig untersuchen, so daß *local search* Verfahren das Problem der Erfüllbarkeit normalerweise nur bei erfolgreicher Suche entscheiden können [CLS98]. Ist aber die Existenz von Lösungen zu erwarten, so können diese mit *local search* oft sehr schnell gefunden werden [CD01]. Besonders bei Constraintproblemen mit relativ vielen Lösungen (im Vergleich zu den möglichen Variablenbelegungen) haben sich *local search* Verfahren als besonders effizient herausgestellt (z.B. [Sol00, CD01]). Allerdings sind die Verfahren oft recht problemspezifisch, weil die Fehlerfunktion, die die unterschiedlichen Weiterentwicklungen vergleicht, besser ist, wenn sie sich auf spezifische Eigenschaften des Problems, anstatt auf ein allgemeines Maß, wie z.B. die Anzahl der verletzten Constraints, bezieht.

Eine zusätzliche Qualität beim Lösen von CSPs mit endlichen Constraintdomänen kann durch Propagation [JL87, Smo95, Apt98] von Constraints eingebracht werden. Verwendet man Propagation, so brauchen nicht alle alternativen Werte für alle Variablen untersucht werden, sondern man kann durch Vorberechnungen die Menge der Werte für jede einzelne Variable einschränken. Im Gegensatz zu *local search* werden dann keine inkonsistenten globalen Zustände verbessert, sondern konsistente Teilzustände solange verfeinert, bis sie global optimale Zustände repräsentieren. Aus der deklarativen Semantik mancher Constraints können bereits bevor seine Variablen instantiiert wurden, Rückschlüsse auf die Variablendomänen durch sogenannte *look ahead* Techniken [Hen89] gemacht werden. Dies wird es in Beispiel 2.2.1 demonstriert wird. Bei vielen Constraints können außerdem sehr effektive Berechnungen von Variablendomänen bzw. Konsistenzprüfungen gemacht werden, wenn eine Variable instantiiert wurde, was in sogenannten *forward checking* Techniken zur Suchraumreduktion ausgenutzt wird [Hen89]. Während der Suche werden die nichtdeterministisch gewählten Werte auf ihre Konsistenz geprüft, indem die Auswirkungen der Instantiierung möglichst weitgehend im Constrainnetzt untersucht werden. Dadurch kann man sehr oft relativ früh feststellen, wenn die getroffene Wertauswahl nicht zu einer Lösung führen kann.

Beispiel 2.2.1 *Das CSP* $(\langle\{1,...,5\},\{1,...,5\},\{1,...,5\},\{1,...,5\}\rangle, \langle a<b, b<c, c<d\rangle)$ *mit dem generierten Constraintschema* $\langle a,b,c,d\rangle$ *kann durch Wissen über die Relation* $<$ *in den ganzen Zahlen zum CSP* $(\langle\{1,2\},\{2,3\},\{3,4\},\{4,5\}\rangle, \langle a<b, b<c, c<d\rangle)$ *durch look ahead reduziert werden. Nimmt man als Constraintdomäne der Variablen die (unendliche) Menge der ganzen Zahlen an, so lassen sich keine Domäneneinschränkungen festlegen.*

Die Veränderung der Domänen der Variablen durch Constraints ist in der CP immer eine Reduktion der möglichen Werte [Apt98]. Daher kann die Propagation von Constraints durch eine Abbildung formalisiert werden, die die Domänen bestimmter Variablen einschränkt.

Definition 2.2.1 (Propagationsfunktion) Seien $D_1,...,D_m$ Constraintdomänen, dann ist $\mathsf{prop} : \mathcal{P}ow(D_1 \times ... \times D_m) \to \mathcal{P}ow(D_1 \times ... \times D_m)$ eine *Propagationsfunktion* wenn $\mathsf{prop}(D) \subseteq D$ gilt, wobei $D \subseteq D_1 \times ... \times D_m$ eine Menge von Variablenbelegungen ist.

Bemerkung 2.2.1
Da Propagationsfunktionen nicht im festen Bezug zu bestimmten Constraints definiert sind, wird an dieser Stelle nicht betrachtet, ob sie die deklarative Semantik eines Constraints erhalten. Propagationsfunktionen haben lediglich die Eigenschaft, die Menge der Eingabevektoren homogen mit der Reduktion der Grundmengen D_i zu reduzieren. Propagationsfunktionen sind daher nicht in jedem Fall domain reduction functions, wie sie in der Literatur oft verwendet werden (z.B. in [Apt98]).

Da Propagationsfunktionen monotone Einschränkungen der Variablendomänen vornehmen und normalerweise idempotent sind, kann man in den meisten Fällen polynomiale Algorithmen finden, die alle Einschränkungen, die sich aus Propagationsmethoden eines CSP ergeben, durchführen. In der Regel führen aber solche Algorithmen nicht zu einer Lösung des implementierten CSP, sondern lediglich zu einer Reduktion der Variablendomänen wie in Bsp. 2.2.1. Das Ziel der Propagation ist also die Abbildung eines CSP auf ein anderes CSP, das "einfacher" ist (z.B. laut [MR98]). Meist bezieht sich diese Eigenschaft auf die Größe des Suchraums, der durch alle Variablen und deren Werte definiert wird. In dem so reduzierten Suchraum kann dann mit *generate-and-test* und *forward-checking* effizienter nach Lösungen gesucht werden. Dabei können Instantiierungen, so wie die anderen Constraints auch, inkrementell verarbeitet werden. Das bedeutet, daß jede Instantiierung zuerst propagiert wird, bevor die nächste Variable instantiiert wird. Mit diesem Vorgehen (*forward checking*) kann man sehr schnell falsche Entscheidungen bei der Wertauswahl für die bereits instantiierten Variablen feststellen [Hen89]. Der Zustand, der durch Propagation in einem Constraintnetz hergestellt wird, hängt von der Reihenfolge und der Häufigkeit der Ausführung der Propagationsfunktionen der Constraints eines gegeben CSP ab. In der Literatur [MR98, Apt98] werden verschiedene Arten der durch Propagation erreichbaren Konsistenz angegeben. Zu jeder Art von Konsistenz sind Algorithmen bekannt, die sie in einem gegebenen CSP erzeugen. Dabei steigt der Rechenaufwand mit zunehmender Aussagekraft des Konsistenzbegriffs und damit meist auch mit zunehmender Suchraumreduktion.

Definition 2.2.2 (Knoten-, arc- und hyper-arc-Konsistenz) Ein Constraint c heißt *Knotenkonsistent*, wenn $\mid \varepsilon(\mathsf{vars}(c)) \mid \neq 1$ oder $\mathsf{vars}(c) = \langle v \rangle \Rightarrow \forall d \in \mathsf{dom}(v) : (d) \in \mathsf{sem}(c)$ gilt. Eine Constraintsequenz C heißt Knotenkonsistent, wenn alle $c \in \varepsilon(C)$ knotenkonsistent sind.

Ein Constraint c heißt *arc-Konsistent*, wenn $\mid \varepsilon(\mathsf{vars}(c)) \mid \neq 2$ oder $\mathsf{vars}(c) = \langle v_1, v_2 \rangle \Rightarrow \forall d_1 \in \mathsf{dom}(v_1) : \exists d_2 \in \mathsf{dom}(v_2) : (d_1, d_2) \in \mathsf{sem}(c)$ gilt. Eine Constraintsequenz C heißt arc-Konsistent, wenn alle $c \in \varepsilon(C)$ arc-Konsistent sind.

Ein Constraint c heißt *hyper-arc-Konsistent*, wenn es für jede Instantiierung einer Variablen $v_i \in \varepsilon(\mathsf{vars}(c))$ mit jedem Wert aus $d_i \in \mathsf{dom}(v_i)$ für jede Variable $v_j \in \varepsilon(\mathsf{vars}(c)) \setminus \{v_i\}$ eine Instantiierung mit einem Wert aus $d_j \in \mathsf{dom}(v_j)$ gibt, so daß $(d_1, ..., d_n) \in \mathsf{sem}(c)$ fur $i, j \in 1..n$ gilt.

Fallstudie, Teil 4
Wenn im Terminplaner $\mathcal{M}$Plan feststeht, daß ein Termin t_1 vor einem Termin t_2 stattfinden soll, so können die Domänen der Constraintvariablen für Beginn und Dauer bereits, ohne konkrete Termine zu kennen, eingeschränkt werden. Diese arc-Konsistenz kann aus dem Constraint "vor" abgeleitet werden:

$vor((tag, zeit, dauer, ort), (tag', zeit', dauer', ort')) :=$
$\{(tag, zeit, dauer, ort, tag', zeit', dauer', ort') \mid vor_{tag}(tag, tag') \vee (tag = tag' \wedge zeit + dauer < zeit')\}$
Zur Herstellung der arc-Konsistenz kann die folgende Propagationsfunktion für das generierte Schema von $\langle vor(t_1, t_2)\rangle$ aufgerufen werden:
$\mathsf{prop}(\mathcal{P}ow(t_1 \times t_2)) := vor(t_1, t_2)$
Bei mehreren Constraints $C = \langle vor(t_1, t_2), vor(t_2, t_3), ..., vor(t_{n-1}, t_n)\rangle$ kann dann durch mehrmaliges Aufrufen von prop *die arc-Konsitenz für C berechnet werden:*
$\forall i \in 1..(n-1) : vor_{tag}(tag_i, tag_{i+1}) \vee (tag_i = tag_{i+1} \wedge zeit_i + dauer_i < zeit_{i+1})$

Der Aufwand zur Herstellung von Knotenkonsistenz liegt in der Klasse $\mathcal{O}(n)$, weil jedes Constraint direkt alle unzulässigen Werte aus der Domäne seiner Variable entfernen kann [MR98]. Um arc-Konsistenz zu erreichen, muß im schlechtesten Fall die Interaktion jedes Constraints mit allen anderen untersucht werden, so daß sich ein quadratischer Aufwand ergibt. Dazu müssen im dann nach der Ausführung einer Propagationsfunktion alle anderen Propagationsfunktionen erneut ausgeführt werden, um die möglichen Einschränkungen in anderen Domänen zu berechnen. In der Praxis kann man aber die Anzahl der erneut aufzurufenden Propagationsfunktionen stark reduzieren, wie es auch in dieser Arbeit beschrieben werden wird. Hyper-arc-Konsistenz kann man im Allgemeinen nur durch exponentielle Algorithmen entscheiden bzw. herstellen, so daß sie in Constraintlösern normalerweise nicht untersucht bzw. hergestellt wird. In manchen Constraintlösern wird die Konsistenz von Constraints über 3 Variablen (die sog. *path*-Konsistenz) [Mon74] erzeugt, die sogenannte k-Konsistenz für Constraints mit $k > 3$ Variablen wird in keinem mir bekannten System hergestellt. Um trotzdem effiziente Propagationsalgorithmen für globale Constraints klassifizieren zu können, wurden andere Arten der Konsistenz für spezielle Constraintdomänen, wie z.B. die *bound*-Konsistenz [MT00] für endliche Mengen ganzer Zahlen, entwickelt. Auf solche Algorithmen möchte ich in dieser Arbeit aber nicht eingehen, sondern setzte voraus, daß sie in den Propagationsfunktionen (Def. 2.2.1) der Constraints und den Definition der Abhängigkeiten in den Variablen (Def. 2.1.2 Punkt 2) implementiert werden können.

Die Eigenschaft der Konsistenz eines CSP bedeutet nicht immer, daß das CSP auch eine Lösung hat, wie in Beispiel 2.2.2 demonstriert wird.

Beispiel 2.2.2 *Das CSP $(\langle\{1,2\}, \{1,2\}, \{1,2\}\rangle, \langle v_1 \neq v_2, v_2 \neq v_3, v_3 \neq v_1\rangle)$ ist arc-Konsistent, hat aber trotzdem keine Lösung, wie sich durch Propagation von Instantiierungen (forward checking) feststellen läßt.*

In manchen Fällen kann durch Herstellung von arc-Konsistenz bereits die Lösung eines CSP gefunden werden. Dies wäre z.B. in Bsp. 2.2.1 der Fall, wenn dort die Domänen im CSP jeweils nur die Werte $\{1, 2, 3, 4\}$ enthielten. In diesem Fall ließe sich durch Propagation die (einzige) Lösung $(1, 2, 3, 4)$ finden. Man kann also ein constraintverarbeitendes System, das arc-Konsistenz herstellt, bereits als Constraintlöser betrachten. Normalerweise ist man aber an Constraintlösern interessiert, die zumindest in den meisten Fällen mindestens eine Lösung finden, wie z.B. die bereits erwähnten *local search* Algorithmen. Außerdem sollte ein Constraintlöser möglichst in der Lage sein, inkonsistente CSPs zu identifizieren, was mit Propagation und Konsistenz-Algorithmen bis zu dem beschrieben Maße erreicht werden kann. Für viele Anwendungen genügt bereits eine dieser Eigenschaften aus, weshalb auch sogenannte unvollständige Constraintlöser verwendet werden. Ist man an allen Lösungen eines CSP interessiert bzw. verlangt man einen Beweis für die Nicht-Existenz von Lösungen, so muß man einen vollständigen Constraintlöser einsetzen. Die *labelling*-Algorithmen, (z.B. [DvHS+88, Hen89]) die z.B. in Constraintlogischen Programmiersystemen wie GNUProlog [Gnu] oder SICStus Prolog [COC97] eingesetzt werden, steuern das Prolog-inhärente *backtracking* so, daß man alle Lösungen zu jedem CSP mit endlichen Constraintdomänen finden kann. Obwohl in solchen vollständigen Constraintlösern nicht nur arc-Konsistenz gefordert wird, werden die Algorithmen zur Herstellung dieser Konsistenz hier normalerweise auch verwendet um den Suchraum a priori einzuschränken und damit die Effizienz des Constraintlösers zu erhöhen.

Definition 2.2.3 (Constraintlöser, vollständiger Constraintlöser) Ist ein CSP (D, C) gegeben, dann liefert ein *vollständiger Constraintlöser* alle Lösungen von C. Ein *(unvollständiger) Constraintlöser* kann eine Lösung von C finden oder feststellen, daß keine existiert.

Weil die Propagation unabhängig von der Suche in Constraintlösern implementiert ist, kann man auch unterschiedliche Suchalgorithmen in einem generischen Constraintlöser verwenden, wie z.B. in JCL [JCL]. Beim asynchronen Constraintlösen können durch die Verwendung unterschiedlicher Suchalgorithmen im Ausführungsmodell sowohl vollständige als auch schnellere unvollständige Constraintlöser definiert werden, was ich in Kapitel 5 beschreiben werde.

2.3 Bekannte Solver und Ausführungsmodelle

Zusammenfassung. In diesem Abschnitt werden die Ausführungsmodelle bekannter Constraintlöser beschrieben. Dabei werden die Eigenschaften besonders herausgehoben, in denen sich das Ausführungsmodell von ACS von den bekannten Modellen unterscheidet. Die Unterschiede werden dann im Verlauf der Arbeit an den entsprechenden Stellen genannt.

Attributierte Variablen und Constraints

Das Konzept der attributierten Variablen [Neu90, Hol92, Hol90] erlaubt das Anhängen von Information an logische Variablen [SS94]. Die meisten Prolog-Systeme, also die Programmiersprachen, die den Einsatz logischer Variablen erlauben, unterstützen inzwischen auch die Attributierung von Variablen [SICS, ecl, Chip, Min]. Attributiert man solche logischen Variablen mit Termrepräsentationen von Variablendomänen im Sinne von Def. 2.1.2 und definiert eine Unifikationsoperation für die Attributwerte, so kann man direkt einen (unvollständigen) Constraintlöser implementieren. Implementiert man zusätzlich eine Prozedur zum *labelling* [Hen89], so erhält man einen vollständigen Constraintlöser für Gleichheitsconstraints, die durch die Unifikation attributierter Variablen implementiert sind. Dazu ist lediglich eine Schnittstelle zur Integration des Unifikationsalgorithmus für die Attribute erforderlich, wie sie z.B. in [Min] oder [ecl] zur Verfügung gestellt wird. Die Möglichkeit, Variablen an Domänen zu binden und diese (z.B. durch Unifikation) einzuschränken, genügt, um mächtige Constraintlöser zu implementieren [CD96]. Auf dieser Basis habe ich auch für die Java-orientierte Prolog-Implementierung Minerva einen Constraintlöser für Variablen mit endlichen Domänen ganzer Zahlen entwickelt [RWG00]. Komplexere Constraints, wie sie in den meisten praktischen Anwendungen verwendet werden, sind aber auf dieser Basis nur ineffizient als Prädikate implementierbar.

Constraintlogische Programmierung

Constraintlogische Programmierung (CLP)[Hen89, JL87, HS88] ist eine Erweiterung der logischen Programmierung [Kow74, SS94], die die Verarbeitung von Constraints in wesentlich weiterem Maße unterstützt als attributierte Variablen. Beide Ansätze sind aber ausschließlich in logischen Programmiersprachen integriert, weshalb ich sie unter dem Begriff CLP zusammenfasse. Zur Zeit sind mehrere kommerzielle und frei erhältliche CLP Systeme erhältlich [SICS, ecl, Chip, Gnu, CLP, JMSY92]. In der CLP wird die Deklarativität und der Nichtdeterminismus der Logischen Programmierung genutzt, wobei die ineffiziente Strategie des *generate-and-test*, die den bei CSP sehr großen Suchraum nur a posteriori einschränkt, ersetzt wird. Der Suchraum wird nicht mehr erst nach dem Entdecken von Inkonsistenzen, sondern durch die Verwendung von Konsistenztechniken (Def.2.2.2) bereits a priori reduziert, wie es im vorherigen Abschnitt beschrieben wurde. Es werden zuerst alle Variablendomänen so weit wie möglich eingeschränkt (*look-ahead*) und dann durch inkrementelle Verarbeitung von nichtdeterministisch definierten Instantiierungen eine Lösung gesucht (*forward-checking*).

Das Ausführungsmodell für constraintlogische Programme ist eine Erweiterung des Modells für logische Programme. Nach jeder Eingabe wird der globale Systemzustand in einer zentralen Datenverwaltung gespeichert, um ihn später wieder herstellen zu können. Dabei können ganze (Teil-) Zustände (sog. *copying*) oder nur Änderungen (sog. *trailing*) abgespeichert werden. Die Vor- und Nachteile dieser Verfahren werden in [Sch99] ausführlich diskutiert. Zusätzlich zur logischen Programmierung beschreibt ein "Zustand" in der CLP auch Constraints und aktuelle Variablendomänen. Die Constraints werden meistens in einem zentralen Constraintspeicher [FA97] verwaltet, während die Variablendomänen (wie Attribute) an logische Variablen angehängt werden. Constraints werden in der CLP inkrementell verarbeitet, d.h. wenn ein Constraint bewiesen werden soll, also zur Ausführung ansteht, so wird es direkt, also synchron mit dem Absetzen des Constraints, im gesamten Zustand propagiert. In den meisten CLP Sprachen bedeutet das, daß arc-Konsistenz hergestellt wird. Entsteht dabei eine Variablendomäne, die keine Elemente mehr hat, so wird ein Fehler erzeugt, der wie in Prolog durch *backtracking* verarbeitet wird. Es wird automatisch nach Alternativen im Suchraum der vorherigen Eingaben gesucht, um den Fehler zu vermeiden. Dazu werden ehemalige Zustände aus der zentralen Datenstruktur abgerufen, um alternative Zweige des Suchbaum zu traversieren. In manchen Prolog Systemen können auch alternative Suchalgorithmen oder Strategien bei der Auswahl der Alternativen angegeben werden.

Constraint Handling Rules

Mit Constraint Handling Rules (CHR) können Constraintlöser durch Regeln definiert werden, die komplexe Constraints durch sogenannte *builtin*-Constraints beschreiben [Frü98]. CHR ist als Programmierbibliothek in manchen CLP Sprachen [SICS, ecl], aber auch als Bibliothek für die objektorientierte Sprache Java [JaCK, Wol01] erhältlich. Durch CHR können in einer deklarativen Syntax spezielle Constraintlöser für konkrete Probleme implementiert werden, die ausschließlich die vorhandenen *builtin*-Constraints und Constraints, die durch Regeln definiert wurden verwenden.

Die Constraints eines CSP werden bei CHR in einem zentralen CHR-Speicher abgelegt, der inkrementell verarbeitet wird. Dazu wird aus dem CHR-Speicher ein Constraint entnommen und durch die bekannten Regeln verarbeitet, wodurch weitere Constraints im CHR-Speicher abgelegt werden. Dazu müssen aus allen bekannten Regeln diejenigen herausgesucht werden, die für das aktuelle Constraint anwendbar sind. *builtin*-Constraints werden nicht durch Regeln verarbeitet, sondern zusammen mit anderen bekannten *builtin*-Constraints vereinfacht. Wird durch diese Vereinfachung eine Inkonsistenz festgestellt oder ist keine Regel mehr anwendbar, so ist ein Finalzustand erreicht. Dies bedeutet, die Propagation der durch Regeln beschriebenen Constraints ist abgeschlossen und die jeweilige Konsistenz damit erreicht. Um Lösungen des CSP in dem so reduzierten Suchraum zu finden, muß die Sprache, in der das CHR-Programm ausgeführt wurde, einen Suchalgorithmus zur Verfügung stellen. Bei Implementierungen in CLP kann hierzu das aus der CLP bekannte *labelling* verwendet werden, wobei die einzelnen Instantiierungen inkrementell durch die CHR verarbeitet werden. In DJCHR [Wol01] wird a priori kein Suchalgorithmus verwendet, so daß man die Möglichkeit hat, unterschiedliche Verfahren zu integrieren. Bei DJCHR ist daher auch keine globale Verwaltung von Zuständen mehr nötig. Allerdings werden in DJCHR alle Constraints in einem globalen Constraintspeicher verwaltet, um das Löschen von Constraints, das zur dynamischen Constraintverarbeitung notwendig ist, zu erlauben. Darauf werde ich in Abschnitt 3.5.2 noch ausführlicher eingehen.

Die CP Bibliotheken von ILOG

Seit Mitte der 90er Jahren vertreibt die französische Firma ILOG [ILOG] sehr erfolgreich C++-Klassenbibliotheken zur Verarbeitung von Constraints. Die Bibliothek zur Bearbeitung von CSPs ILOG Solver [Ilog] und die neue Java-Version ILOG JSolver [JSol] sind Weiterentwicklungen des CLP Schemas [Pug94]. Allerdings wurde das ursprüngliche Modell durch die Objektorientierung und aus softwaretechnischen Gründen inzwischen stark modifiziert. Der wesentliche Unterschied zu CLP Systemen besteht im ILOG Solver 5.0 darin, daß Constraints nicht inkrementell verarbeitet werden. Constraints werden in ein sogenanntes Modell eingefügt (oder wieder daraus entnommen), ohne das dabei Propagation stattfindet. Ein so definiertes CSP kann dann mit unterschiedlichen Suchverfahren gelöst werden. In der Bibliothek werden dazu unvollständige *local search* Algorithmen und vollständige Verfahren wie *labelling* mit *backtracking* zur Verfügung gestellt. In jedem Fall liefert der Constraintlöser aber nur geschlossene Lösungen und keinen Zustand, in dem eine Art von Konsistenz hergestellt wurde und die Variablen noch nicht instantiiert sind. Details über das Ausführungsmodell von ILOG Solver sind nur rudimentär zugänglich. Das anfängliche Ziel, den CLP Ansatz für eine objektorientierte Sprache anzupassen [Pug94], wurde inzwischen aber von der Philosophie, konkrete Werkzeuge und Constraints für wirtschaftlich relevante Anwendungsbereiche herzustellen, abgelöst.

Constraintprogrammierung mit Mozart und Figaro

In der objektorientierten, funktional-logischen Programmiersprache Oz [MOz] wurde von Anfang an die Verarbeitung von Constraints unterstützt (z.B.[HW96]). In der aktuellen Version 1.2.3 der Oz-Programmierumgebung Mozart sind Constraints über endliche Domänen ganzer Zahlen, endliche Mengen [DvHS+88] und rationale Bäumen [Cou83] vorhanden. Neue Arten von Constraints und Constraintdomänen können durch eine Schnittstelle integriert werden. Im Gegensatz zu allen anderen vorgestellten Ausführungsmodellen wird in Mozart auch die Bearbeitung von verteilten CSPs unterstützt.

Constraints werden in Mozart in monolithischen Anwendungen inkrementell verarbeitet und durch Propagation wird arc-Konsistenz zur Reduktion des Suchraums synchron zum Absetzen der Constraints wie bei CLP hergestellt. In verteilten CSPs wird allerdings Propagation nur im Sinne von *forward checking* unterstützt, es wird also arc-Konsistenz zwischen Variablen unterschiedlicher Agenten nur dann hergestellt, wenn eine der Variablen instantiiert wurde. In Mozart können im Unterschied zu CLP gekapselte Suchalgorithmen [SSW94, HvRBS98] verwendet werden, die nicht auf einem globalen Zustandsraum arbeiten. Die Algorithmen sind austauschbar, so daß vollständige und unvollständige Constraintlöser durch Verwendung unterschiedlicher Suchalgorithmen erzeugt werden können. Die Kapselung der Suchalgorithmen und damit der Verzicht auf die zentrale Verwaltung ehemaliger Zustände erlaubt die verteilte Anwendung, so daß die für Mozart entwickelten Algorithmen zur Wiederherstellung ehemaliger Zustände auch in der neuen Programmbibliothek für verteilte CSPs Figaro [Fig, TYC00] verwendet werden.

Die Idee der gekapselten Suche besteht darin, den Suchraum explizit zu verwalten [Sch00] und ihn nicht implizit durch die Verwaltung ehemaliger globaler Zustände und Alternativen wie in Prolog zur Verfügung zu stellen. Da in den meisten Fällen allein durch Propagation weder eine Lösung eines CSP, noch eine Inkonsistenz entdeckt werden kann, ist es oft auch nicht erforderlich, die Zustandsveränderungen während der inkrementellen Verarbeitung des CSP zu verfolgen. Dadurch kann der implizite Suchraum einer CLP Anwendung auf den Suchraum, der durch die Anwendungen eines nichtdeterministischen Algorithmus entsteht, reduziert werden. Für solche reduzierten Suchräume werden in Mozart und auch Figaro zustandsbasierte Traversierungsalgorithmen zur Verfügung gestellt, d.h. die Traversierung des Suchbaums kann auf unterschiedliche Weise (z.B. *depth-first* oder *breadth-first*) bewerkstelligt werden, wobei der Suchraum selbst nicht modifiziert wird. Dazu werden auch in Mozart oder Figaro ehemalige (gekapselte) Zustände wiederhergestellt und alternativ weiterentwickelt. Zur Wiederherstellung ehemaliger Zustände können *trailing*- oder *copying*-Techniken wie bei CLP verwendet werden, die den Suchbaum von den Blättern aus zurückverfolgen. Alternativ werden aber auch Techniken untersucht, die ehemalige Zustände von der Wurzel des Suchbaums aus rekonstruieren [CHN01].

Constraints als explizite Relationen

In der Java Constraint Library [JCL] und im Relational Constraint Solver (RCS) [See01], der gegenwärtig bei DaimlerChrysler entwickelt wird, werden Constraints explizit als Relationen implementiert. Es werden also alle zulässigen Variablenbelegungen für jedes Constraint abgespeichert. Da dies zu sehr großen Datenmengen führt, können in JCL von vorneherein nur binäre Constraints verarbeitet werden, während in RCS versucht wird, diese Datenmengen bei der Konjunktion von Constraints zu verkleinern. Beide Solver sind aber in ihrer Laufzeit nicht mit anderen bekannten Systemen vergleichbar (vgl. Abschnitt 7.3). Diese Systeme sind daher nur für akademische Zwecke bzw. für einfache CSPs geeignet.

2.4 Chaotische Iteration

Zusammenfassung. Die für ACS relevanten Ergebnisse der Anwendung der Theorie der Chaotischen Iteration auf die Propagation in der CP aus [Apt98] wird kurz zusammengefaßt und für meine spezielle Anwendung konkretisiert. Durch die Verwendung von Mengen als Domäne von Propagationsfunktionen können strukturelle Eigenschaften der Funktionen abgeleitet werden, die die Existenz eines eindeutig bestimmten Fixpunktes des Iterationsprozeß sichern.

Chaotische Iteration zur Berechnung von Grenzwerten der zufälligen und mehrfachen Ausführung endlicher Mengen von Funktionen wird seit den 60er Jahren im Bereich der Numerik untersucht (z.B. [CM69]). In endlichen Constraintprogrammen können solche Mengen von Funktionen aus den Constraints abgeleitet werden. Die Propagation der Constraints kann dann als chaotischer Iterationsprozeß betrachtet werden [Apt98], was die mathematischen Ergebnisse anwendbar für die Constraintprogrammierung macht (z.B. [MR99, Mon00]). Beim asynchronen Constraintlösen ist der Propagationsprozeß der Constraintverarbeitung wegen der Asynchronität chaotisch, so daß sich auch hier eine Anwendung der Theorie ergibt. In meinem Ansatz wird eine Menge von Funktionen aus einem beliebigen Constraintprogramm abgeleitet, auf der sich eine Chaotische Iteration durchführen läßt. Das Ergebnis der Iteration betrachte ich als intendierte Semantik des Programms, die es operational zu berechnen gilt. In diesem Abschnitt werde ich die Theorie aus [Apt98] für diese Verwendung zusammenfassen und konkretisieren.

Definition 2.4.1 (Chaotische Iteration) Sei eine Menge D und eine Menge $F = \{f_1, ..., f_n\}$ von Funktionen in D gegeben, dann ist die *chaotische Iteration* F^+ eines Wertes $d \in D$ eine unendliche Sequenz von Werten $d_i \in D$ mit $d_0 := d$ und $d_j := f_{i_j}(d_{j-1})$. Dabei muß gelten, daß jedes $f_i \in F$ unendlich oft angewandt wird.

Ziel der Propagation in der CP ist die Einschränkung des Suchraums für das implementierte CSP. Diese Einschränkung wird dabei durch die Einschränkung von Variablendomänen erreicht. Um den Begriff der Einschränkung zu formalisieren bietet es sich an, eine Ordnung auf der Menge aller bekannten Variablendomänen anzunehmen, so daß sich für die Propagationsfunktionen notwendige Eigenschaften definieren lassen.

Definition 2.4.2 (Funktionen in Ordnungen) Sei eine partielle Ordnung $(D, \sqsubseteq)$ gegeben, dann heißt eine Funktion $f : D \rightarrow D$

- inflationär, wenn $x \sqsubseteq f(x)$ für alle $x \in D$ gilt
- monoton, wenn $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ für alle $x, y \in D$ gilt

Weil Variablendomänen laut Definition 2.1.2 und 2.1.1 in dieser Arbeit immer als Mengen interpretiert werden können, gebe ich die konkrete Ordnung der Mengeninklusion an, in der die Propagationsfunktionen inflationär sind.

Lemma 2.4.1 Gegeben sei ein Constraintschema $\langle v_1, ..., v_n \rangle$. Dann ist eine Propagationsfunktion in $\mathcal{P}ow(D) = \mathcal{P}ow(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n))$ inflationär bzgl. $(\mathcal{P}ow(D), \supseteq)$.

Beweis:
Folgt direkt aus Def. 2.2.1.

□

Die Monotonie von Propagationsfunktionen bzgl. der Mengeninklusion ergibt sich aus der Monotonie der Kreuzproduktbildung. So daß ich für Propagationsfunktionen die Monotonie allgemein zeigen kann.

Lemma 2.4.2 Gegeben sei ein Constraintschema $\langle v_1, ..., v_n \rangle$. Dann ist eine Propagationsfunktion in $\mathcal{P}ow(D) = \mathcal{P}ow(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n))$ monoton bzgl. $(\mathcal{P}ow(D), \supseteq)$.

Beweis:
Das Lemma wird indirekt bewiesen. Seien $M_1, M_2 \subseteq \mathcal{P}ow(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n))$ mit $M_1 \subseteq M_2$ gegeben. Angenommen, es gilt für eine beliebige Propagationsfunktion f: $f(M_1) \supset f(M_2)$, dann muß ein Wert d in einer Domäne $\mathsf{dom}(v_i)$ existieren, der in $f(M_1)$ aber nicht in $f(M_2)$ enthalten ist. Wegen $M_1 \subseteq M_2$ muß d aus M_2 durch f entfernt worden sein, dann kann er aber auch nicht in $f(M_1)$ enthalten sein.

□

Funktionen, die im Einklang mit der deklarativen Semantik der Constraints eines CSP Domäneneinschränkungen definieren, werden in [Apt98] als Domänenreduktionsfunktionen (*domain reduction functions*) charakterisiert. Die chaotische Iteration solcher Funktionen dient dort als Modell des Propagationsprozesses bei der Constraintprogrammierung. Ich schränke die allgemeine Definition aus [Apt98] zur Verwendung in ACS ein, indem ich voraussetze, daß Domänenreduktionsfunktionen immer auch Propagationsfunktionen sind. Dadurch kann ich die Domäne der Funktionen von einer beliebigen Ordnung zur Mengeninklusion konkretisieren.

Definition 2.4.3 (Domänenreduktionsfunktion) Eine Propagationsfunktion prop für ein Constraint c heißt *Domänenreduktionsfunktion*, wenn $\mathsf{sem}(c) \cap D = \mathsf{sem}(c) \cap \mathsf{prop}(D)$ für jede beliebige Menge von Variablenbelegungen D gilt.

Bemerkung 2.4.1
Durch die Anforderung wird sichergestellt, daß bei der Ausführung von prop *keine Lösung verlorengeht, denn es gilt insbesondere auch* $\mathsf{sem}(c) \subseteq D \Rightarrow \mathsf{sem}(c) \subseteq \mathsf{prop}(D)$.

Ein zentrales Ergebnis aus [Apt98] ist der Domänenreduktionssatz, der sicherstellt, daß die chaotische Iteration unter bestimmten Voraussetzungen den Propagationsprozeß in der CP charakterisiert. Unter der Annahme, daß Constraints monotone Domänenreduktionsfunktionen definieren, führt die unendlich häufige Ausführung dieser Funktionen zu einem eindeutigen Ergebnis.

Satz 1 (Domänenreduktion) Gegeben sei ein CSP $P = (\langle D_1, D_2, ..., D_n\rangle, C)$ so daß für alle D_i eine partielle Ordnung mit Infimum definiert ist. Sei weiter die Menge $F = \{f_1, ..., f_m\}$ gegeben, wobei jedes f_i die Domänenreduktionsfunktion für ein Constraint aus C ist. Wenn alle f_i monoton bzgl. der Mengeninklusion sind, gilt:

- Der Grenzwert der chaotischen Iteration F^+ des Wertes $D_1 \times ... \times D_n$ existiert
- der Grenzwert ist $\widehat{F} = \bigcap_{j=0..\infty} f^j(D_1 \times ... \times D_n)$, wobei f definiert ist durch $f(D) = \bigcap_{i=1..m} f_i^+(D)$.

Bemerkung 2.4.2
Domänenreduktionsfunktionen sind in meinem Ansatz laut Def. 2.4.3 ausschließlich Propagationsfunktionen, so daß die Monotonie laut Lemma 2.4.2 gesichert ist. Die Funktionen bilden in meinem Ansatz Mengen von Variablenbelegungen ab, so daß die erforderliche partielle Ordnung durch $(\mathcal{P}ow(D), \supseteq)$ *in den Lemmata 2.4.1 2.4.2 angegeben wurden. Das Infimum der Ordnung ist demnach* $\emptyset$. *Damit ist der Satz für Domänenreduktionsfunktionen wie sie in Def. 2.4.3 definiert wurden anwendbar.*

Bemerkung 2.4.3
Der Grenzwert der chaotischen Iteration zusammen mit den Constraints C aus Satz 1 ergeben ein CSP, das äquivalent zu P ist. Die Variablendomänen sind aber in der Regel stark eingeschränkt.

Der Domänenreduktionssatz sichert im Zusammenhang mit asynchronem Constraintlösen die Existenz einer denotationalen Semantik von Constraintprogrammen. Ich werde in Kapitel 3 zeigen, wann Constraints beim ACS monotone Domänenreduktionsfunktionen definieren und somit, wann ein Ergebnis der Iteration existiert. Dieses Ergebnis, das im unendlichen Iterationsprozeß berechnet wird, soll auch durch die endliche Ausführung von Constraintimplementierungen in meinem Ausführungsmodell erreicht werden.

Eine wichtige Folgerung aus der Arbeit [Apt98] ist, daß die Reihenfolge in der Domänenreduktionen durchgeführt werden, irrelevant für das Ergebnis ist. Es muß dazu lediglich vorausgesetzt werden, daß:

1. die verwendeten Einschränkungen sich richtig (inflationär und monoton) verhalten,
2. daß die Menge der Funktionen endlich ist und

3. daß jede Funktion unendlich oft ausgeführt wird.

Die Idempotenz der Domänenreduktionsfunktionen, die in der Literatur oft vorausgesetzt wird (z.B. [MS98]), ist keine notwendige Voraussetzung für ein eindeutiges Ergebnis [Apt98]. Die erste Anforderung wird beim ACS durch die Möglichkeiten zur Implementierung von Constraints sichergestellt. Die zweite Anforderung (endlich viele Funktionen) ist in manchen Anwendungen, wie z.B. der Terminplanung aus Abschnitt 1.3, nicht direkt gegeben, weil dort ein kontinuierliches Problem, also potentiell unendlich viele Constraints, bearbeitet wird. Um in solchen Anwendungen die formale Semantik abzuleiten, kann man das kontinuierliche Problem auf ein abgeschlossenes abbilden, das alle Constraints enthält, die vom Programmstart bis zu einem festen Zeitpunkt abgesetzt wurden. Das Ergebnis eines asynchronen Constraintlöseprozesses kann also immer nur als Zustand im Zusammenhang mit vorangegangenen Eingaben angegeben werden. Die dritte genannte Voraussetzung für die Anwendbarkeit der Konstruktion der Semantik durch chaotische Iteration, nämlich daß alle Funktionen unendlich oft ausgeführt werden, kann durch einen *fairen* Propagationsprozeß simuliert werden. Im Ausführungsmodell muß sichergestellt werden, daß jede Propagationsfunktion mindestens so oft ausgeführt wird, daß sie keine weiteren Domäneneinschränkungen mehr ableiten kann.

Kapitel 3

Asynchrones Constraintlösen

In diesem Kapitel wird das von mir entwickelte Ausführungsmodell des asynchronen Constraintlösens definiert. Ich gebe einige Leistungsmerkmale für Constraintlöser in komplexen Softwaresystemen an, durch die sich ACS von anderen Ausführungsmodellen abgegrenzt. Constraintprogramme werden in ACS als Erweiterungen imperativer Programme um Anweisungen zur Constraintverarbeitung definiert. Die Ausführung solcher Programme berechnet Variablendomänen aus asynchron abgesetzten oder zurückgenommenen Constraints. Wenn alle abgesetzten Constraints wohldefiniert sind, so kann aus einem Constraintprogramm ein eindeutiges Ergebnis berechnet werden. Dieses Ergebnis entspricht der denotationalen Semantik des Programms, die ich als den Fixpunkt der chaotischen Iteration, der durch die Constraints definierten Domänenreduktionsfunktionen betrachte.

Durch die in Abschnitt 2.4 beschriebene Tatsache, daß die Reihenfolge, in der Constraintpropagationsmethoden ausgeführt werden, unerheblich für das Ergebnis der Domäneneinschränkungen ist, entsteht die Möglichkeit, ein nichtdeterministisches Ausführungsmodell zur Constraintpropagation zu definieren. Durch die Möglichkeit, die Abarbeitungsreihenfolge beliebig zu variieren, kann sich die Implementierung von Constraintlösern mehr an ihrem gewünschten Verhalten als Softwarekomponente als an einen bestimmten Ausführungsmodell (Abschnitt 2.3) orientieren. Durch den Verzicht auf eine deterministische Abarbeitung der Propagationsmethoden kann beim asynchronen Constraintlösen allerdings keine zustandsbasierte Suche, wie in fast allen bekannten Constraintlösern, in das System integriert werden. Um trotzdem NP-vollständige Probleme, deren Lösung nicht-deterministische Suche erfordert, bearbeiten zu können, muß eine Rücksetzoperation definiert werden, die getroffene Entscheidungen rückgängig macht. Beim asynchronen Constraintlösen wird diese Rücksetzoperation nicht wie in den meisten bekannten Ausführungsmodellen durch das rückwärts- oder vorwärtsgerichtete Wiederherstellen von Zuständen von den Blättern oder der Wurzel des Suchbaums aus, sondern durch Berechnung des gewünschten Zustands aus lokal gespeicherten Daten ausgeführt. Anstatt die

Zustände des Problemlöseprozesses zu betrachten, wird also eine "Umkehroperation" zum Absetzen von Constraints zur Verfügung gestellt.

3.1 Leistungsmerkmale von Constraintlösern

Zusammenfassung. Bei der Verwendung von Constraintlösern als Komponente von komplexen Softwaresystemen, sollten sie als abstrakter Datentyp die Möglichkeit bieten, Constraints abzusetzen und die dadurch entstehenden CSPs zu lösen. In diesem Abschnitt werden einige Eigenschaften angegeben, die eine solche *black-box*-Komponente haben sollte, um gut in konkreten Anwendungen eingesetzt werden zu können. Aus der Beschreibung des gewünschten Verhaltens werden Rückschlüsse auf die interne Realisierung von Constraintlösern gemacht.

Die Eignung und Angemessenheit von constraintbasierten Methoden in verschiedenen Anwendungsbereichen der Künstlichen Intelligenz wurde bereits in der Einleitung diskutiert. So bietet sich die Verwendung von constraintbasierten Problemlösekomponenten in komplexen Softwaresystemen an, die *unter anderem* z.B. Funktionen zur Planung oder Konfiguration zur Verfügung stellen sollen. Durch die Eigenschaften und Benutzbarkeit vieler bekannter Constraintlöser wird eine solche Verwendung als eine Komponente eines komplexen Systems aber nicht besonders gut unterstützt. Es ist vielmehr oft nötig, eine spezielle Implementierungssprache zur Constraintverarbeitung zu verwenden (z.B. CHIP in [JG00, GM00]), die in vielen Fällen Defizite z.B. bei der Anbindung an Datenbanken, der Unterstützung von Nebenläufigkeit und Verteilung o.ä. hat.

Als Implementierungssprache für das neue Ausführungsmodell habe ich daher die Sprache Java gewählt, die durch ihre weite Verbreitung Schnittstellen oder APIs für alle gängigen Softwarekomponenten zur Verfügung stellt. Im Unterschied zur Entwicklung von Mozart/Oz [MOz] sollte keine neue Sprache definiert werden, sondern ein Programmierbibliothek zur Verfügung gestellt werden, die von Java Programmierern leicht zu verstehen und zu verwenden ist. Die Herstellung einer Bibliothek für eine verbreitete Programmiersprache war auch bei der Konzeption der C++ Bibliothek Figaro [Fig] beabsichtigt, die ansonsten die Konzepte von der Sprache Mozart [MOz] weiterverfolgt, aber versucht, dadurch einen neuen Benutzerkreis zu erschließen.

Die Eigenschaft, daß ein Paket einfach zu verstehen oder zu verwenden ist, kann sicherlich nur sehr subjektiv aus der Sicht des jeweiligen Benutzers beurteilt werden. Die im folgenden aufgezählten Leistungsmerkmale von Constraintlösern sind daher auch eher technisch motiviert, ich habe die Eignung einer Constraintlöser-Softwarekomponente unter Berücksichtigung bestimmter gewünschter Eigenschaften moderner Softwaresysteme dargestellt. Die Benutzung zielt dabei auf die Verwendung des Constraintlösers als gekapselte Wissensverwaltung ab. In Abbildung 3.1 wird die intendierte Handhabung an meiner durchgängigen Fallstudie dargestellt.

Die genannten Leistungsmerkmale sollen zur Identifikation *besonderer* Eigenschaften von Constraintlösern dienen, anhand derer sich bestehende Constraintlöser unterscheiden lassen. Ich werde nicht auf die in der Literatur oft untersuchten Eigenschaften des Propagations- oder Suchprozesses von Cons-

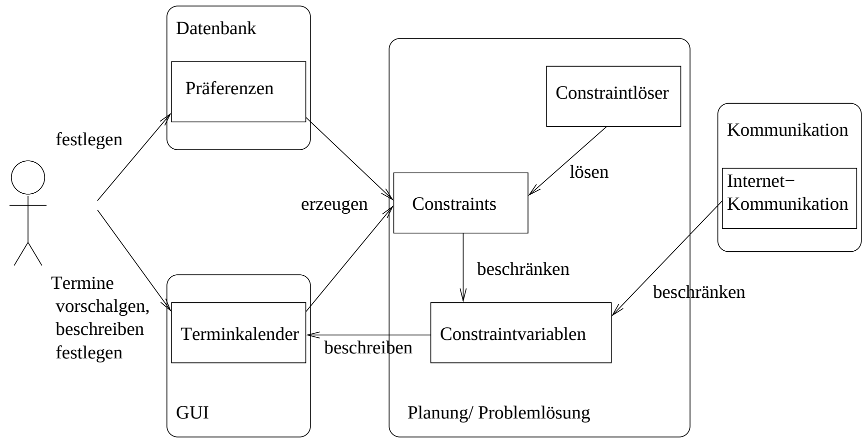

Abbildung 3.1: **Fallstudie Teil 4**: Benutzung und wesentliche Komponenten von $\mathcal{M}$Plan

traintlösern eingehen. Eigenschaften, die durch die Constraintdomäne des Solvers gegeben sind werden also nicht betrachtet (z.B. Konsistenztest, Simplifikation, Projektion oder Entailment). Als Basis der folgenden Diskussion betrachte ich Propagationsfunktion, die von Constraints zur Verfügung gestellt werden und durch Einschränkung von Variablendomänen eine bestimmte Semantik operational umsetzen. Es wird vorausgesetzt, daß diese Methoden die o.g. gewünschten Eigenschaften von Constraintlösern implementieren.

Effizienz

Dem Zeit- und Speicheraufwand von Constraintlösern wurde in der CP von jeher große Aufmerksamkeit gewidmet. Dies ergibt sich natürlich aus der Art der Probleme, die mit constraintbasierten Methoden vorwiegend bearbeitet werden. Es sind schnelle Werkzeuge notwendig, um große NP-vollständige Probleme überhaupt in vertretbarer Zeit lösen zu können. Für die Verwendung in Constraintverarbeitenden Systemen ergibt sich daher die Notwendigkeit, die Berechnung der Propagation und die der Suche so zusammenspielen zu lassen, daß sich die Probleme insgesamt möglichst schnell lösen lassen. Das Ausführungsmodell selbst sowie die Anwendung in der der Constraintlöser verwendet wird, sollen dabei eine möglichst geringe Rolle spielen, d.h. das Aufrufen von Propagationsfunktionen, die Instantiierung von Variablen und die Verarbeitung von Ein- und Ausgaben soll in linearer Zeit erledigt werden.

Es hat sich gezeigt, daß es kein Patentrezept für die richtige Mischung von Propagation und Suche gibt. In den meisten Constraintlösern (z.B. [SICS, MOz]) wird daher z.B. bei bestimmten Constraints arc-Konsistenz durch Propagation hergestellt und bei anderen der Berechnungsaufwand auf die Suche verschoben. Manche Constraintlöser wie z.B. ILOG [ILOG] erlauben sogar, Probleme völlig ohne Propagation unter Verwendung von besonders effizienten Suchverfahren zu lösen. Dies ist sicherlich für bestimmte Probleme sehr effizient, aber für andere, die z.B. sehr wenige Lösungen haben, ungeeignet. Eine individuelle Gestaltung vom Aufwand der Propagation im Verhältnis zur Suche ist also eine wichtige Möglichkeit, Constraintlöser effizient arbeiten zu lassen. Es sollte für jede Constraintdomäne oder evtl. sogar für jede Anwendung individuell bestimmbar sein, wieviel Propagation bei welchem Constraint durchgeführt wird.

Bei der Betrachtung des Rechenaufwands constraintbasierter Anwendungen wird in der Regel die Zeit zum Finden von Lösungen ungeachtet von anderen Berechnungen des Softwaresystems betrachtet. Es werden bestimmte Benchmarktests (z.B. [GW]) zum Lösen bestimmter kombinatorischer Spiele, wie z.B. das berühmte n-Damen Problem, untersucht, um die Leistungsfähigkeit des Constraintlösers zu beurteilen. In der Praxis kann die Anwendung eines nach solchen Maßstaben effizienten Solvers aber sehr langsam sein, wenn z.B. die Darstellung der Ergebnisse sehr ineffizient ist. Es ist also geboten, einen Solver so zu gestalten, daß seine Effizienz zum Lösen des Constraintproblems nicht durch den Aufwand anderer Softwarekomponenten gebremst wird.

Da die Reihenfolge der Ausführung von Propagationsfunktionen, wie beschrieben, keine Rolle spielt und die Suchverfahren zum Lösen von CSPs ohnehin in der Regel nicht-deterministisch sind, ergeben sich Freiheitsgrade, die die Effizienz des Constraintlösers stark beeinflussen. Es wurden in der Vergangenheit sehr viele Untersuchungen von Variablen- bzw. Wertauswahlheuristiken in Suchalgorithmen gemacht [Hen89]. Dabei hat sich gezeigt, daß eine günstige Festlegung bzw. dynamische Anpassung der Reihenfolgen hier enorme Effizienzgewinne bringen kann. Ein effizienter Constraintlöser sollte also verschiedene Reihenfolgen zur Ausführung von Instantiierungen in Suchalgorithmen erlauben. Weiterhin kann auch durch die Reihenfolge der Ausführung von Propagationsfunktionen ein Performanzgewinn erzielt werden. Daher erlaubt das ECL^{i}PS^e-System [WNS97] von jeher das Ordnen von Propagationsfunktionen vor ihrer Ausführung. Es ist oft geboten, neues Wissen möglichst früh in einen Lösungsprozess einfließen zu lassen, um keine unnötigen Berechnungen unter Annahme veralteten Wissens machen zu müssen. So ist z.B. sicherlich das Zurücknehmen eines Constraints immer so früh wie möglich zu erledigen, damit das zurückgenommene Constraint noch möglichst wenig Auswirkungen im Constraintnetzwerk hat. Aber auch bei Propagation ohne Constraintrücknahme kann man durch eine geschickte Reihenfolge der Ausführung von Propagationsfunktionen viel Zeit gewinnen. In Kapitel 7.1 werden wir z.B. ein Problem vorstellen, dessen Propagationsaufwand sich durch gute Sortierung von $\mathcal{O}(n^2 \log n)$ auf $\mathcal{O}(n)$ reduzieren ließ.

Inkrementalität

Wenn ein Constraintlöser neue Constraints so verarbeiten kann, daß ihre intendierte Semantik operational so weit wie möglich bzw. gewünscht umgesetzt wird, ohne alle bisherigen Berechnung erneut auszuführen, so nennt man ihn inkrementell. In inkrementellen Constraintlösern werden also bereits ausgeführte Berechnungen auch bei Erweiterungen der Wissensbasis beibehalten und berücksichtigt. In der Literatur (z.B. [FA97]) wird Inkrementalität oft als wichtige Voraussetzung für Effizienz von Constraintlösern angegeben. Die Ausführung von Propagationsfunktionen erzeugt direkt alle aus dem Constraint ableitbaren Domäneneinschränkungen, wobei Propagationsfunktionen nur dann erneut ausgeführt werden, wenn dies zu weiteren Domäneneinschränkungen führen kann.

Die Effizienz ergibt sich hierbei aus der Wiederverwendung von Berechnungen.

Analog zum Begriff der Inkrementalität bzgl. des Hinzufügens von Information durch neue Constraints kann auch das Zurücknehmen von Constraints als Informationsgewinn betrachtet werden. Daraus ergibt sich als Anforderung an einen inkrementellen adaptiven Constraintlöser, daß das Entfernen eines Constraints zu dem Zustand führt, der entstanden wäre, wenn das Constraint nie abgesetzt worden wäre. Dieser Zustand soll dabei nicht neu aus allen bekannten Constraints berechnet werden, sondern es sollen möglichst große Teile bereits ausgeführter Berechnungen wiederverwendet werden. In constraintverarbeitenden Systemen, die frühere Zustände in Ketten von Umformungen, also durch sogenanntes *trailing* (z.B. [Sch99, CD96, AB93, Hen89]) abspeichern, bedeutet das, daß die Berechnungen ab einem bestimmten Zustand neu ausgeführt werden. Es ist aber auch möglich, die richtigen Zustände aus dem gegebenen Constraintnetz direkt zu berechnen (z.B. [Sch97, CHN01, Rin01c]), ohne dabei alle Constraints erneut ausführen zu müssen.

In manchen Anwendungsgebieten hat sich inzwischen gezeigt, daß Inkrementalität keine unbedingte Voraussetzung zum schnellen Finden von Lösungen für manche CSPs ist [CD01], so daß der Wunsch nach Effizienz nicht mehr unbedingt die Verwendung von inkrementellen Constraintlösern erfordert. Reine Suchverfahren, die Variablenbelegungen generieren und ihre Richtigkeit überprüfen und mit guten Heuristiken erzeugte Variablenbelegungen in Richtung einer Lösung verändern, wie die in Abschnitt 2.2 beschriebenen *local search*-Verfahren, finden oft schneller eine Lösung für a-priori bekannte Probleme. Allerdings sind solche Verfahren nicht zur Verwendung in Anwendungen geeignet, in denen neues Wissen z.B. durch Benutzerinteraktion nach und nach gesammelt wird. Bei der Anwendung von Planungswerkzeugen [GM00] hat sich in der Praxis oft gezeigt, daß es nicht möglich ist, aus einer geschlossenen Darstellung allen Wissens eine Lösung zu generieren, weil eine solche Darstellung nicht existiert. Benutzer sind oft nur dann in der Lage, ihr Domänenwissen detailliert genug anzugeben, wenn sie vor konkrete Fragen gestellt werden. Solche konkrete Fragen entstehen bei der inkrementellen Verarbeitung vorheriger Eingaben aus den noch undefinierten Variablen und ihren Domänen.

In kontinuierlich auftretenden Constraintproblemen, wie z.B. der Terminplanung aus meiner Beispielanwendung, können ebenfalls keine reinen Suchverfahren zur Lösungsfindung verwendet werden. Dies ergibt sich aus der Tatsache, daß zu keinem Zeitpunkt eine Lösung des gesamten CSP gesucht wird. Es werden ständig neue Constraints erzeugt und abgesetzt, was zu Einschränkungen von Variablendomänen führt, aber immer nur von bestimmten Variablen verlangt, daß sie instantiiert sind. Daraus ergibt sich, daß jedes neue Constraint so verarbeitet werden muß, daß möglichst weitgehende Einschränkungen gemacht werden, um die Existenz einer Lösung möglichst immer zu erhalten. Für den Constraintlöser bedeutet diese Anforderung, daß er wenn möglich inkrementell hyper-arc-Konsistenz sichern sollte.

Adaptivität

Betrachtet man abgesetzte Constraints als Basis für das durch Propagation ableitbare Wissen, so sollte sich dieses Wissen stets exakt auf aktualisierbare Mengen gültiger Constraints beziehen. Insbesondere in interaktiven Anwendungen können neue Constraints oft hinzugenommen oder entfernt werden, so daß sich das Basiswissen und damit auch das ableitbare Wissen stets ändert. Ein Constraintlöser sollte auf Veränderungen der Wissensbasis (nämlich der Constraints) immer so reagieren, daß nicht mehr und nicht weniger Domäneneinschränkungen an Variablen bestehen, als sich aus den Propagationsfunktionen der Constraints ableiten lassen. Der Constraintlöser fungiert als *thruth-maintainance-system* [Doy79], indem er alle Variablendomänen ständig an die aktuell gültigen Constraints anpaßt.

Aus der Sicht des Programmierers einer constraintbasierten Anwendung bedeutet Adaptivität, daß er jederzeit neue Constraints in die Wissensbasis einfügen und bekannte Constraints entfernen kann. Das "Wissen" betrachtet er dabei als Datentyp, der die genannten Operationen zur Verfügung stellt und in dem inferiertes Wissen abgelesen werden kann. Da die interne Verarbeitung der Constraints, also die Ausführung der Operationen Hinzufügen und Löschen bei diesem Gebrauch des Datentyps keine Rolle spielt, sollten sich hieraus auch keine Anforderungen an den Programmierer ergeben. Das heißt, eine constraintverarbeitende Komponente sollte als *black box* die genannten Operationen zur Verfügung stellen und als Ausgabe jederzeit alles aus den abgesetzten Constraints ableitbare Wissen durch aktuelle Variablendomänen zur Verfügung stellen.

Asynchronität

Zur inkrementellen Bearbeitung von verteilten CSPs lösen mehrere Constraintlöser kooperativ ein gemeinsames Problem [Han01]. Wird dabei die inferierte Information über Variablen kommuniziert [BN95], so veranlassen Variablen oft die erneute Ausführung von Propagationsfunktionen in anderen Agenten. Da die Kommunikation an sich schon recht ineffizient sein kann und weil zur Vermeidung von Leser/Schreiber- Problemen solche Zugriffe synchronisiert werden müssten, ist Asynchronität ein wichtiges Leistungsmerkmal für Solver in verteilten Anwendungen. Wenn die Operationen, die zur Kooperation der Solver notwendig sind (Constraint hinzufügen und zurücknehmen) auch asynchron ausgeführt werden können, genügt zur Kommunikation ein nicht-blockierender Nachrichtenkanal. Damit muß zur Kommunikation während der Problemlösung lediglich eine Nachricht asynchron verschickt werden, ohne ein Ergebnis abzuwarten. In Kapitel 6 werde ich noch genauer auf die Konzepte der Verteilung und die Lösungen, die sich durch asynchrones Constraintlösen ergeben, eingehen.

Bei der Benutzung von Constraintverarbeitenden Komponenten in komplexen Softwaresystemen ist es oft nicht nötig, das Ergebnis der Propagation neuer Eingaben direkt zu verwenden. Insbesondere bei interaktiv entstehenden oder kontinuierlichen Problemen kann die Propagation neuen Wissens im Hintergrund ablaufen und braucht nur in bestimmten Situationen abgeschlossen zu sein. So kann die Zeit, in der z.B. ein Anwender Eingaben macht oder in der mit anderen Softwareagenten kommuniziert wird, zum Lösen von vorher definierten Problemen genutzt werden. Wenn das Absetzen und Zurücknehmen von Constraints in einem Constraintlöser auch asynchron möglich ist, können solche unabhängigen Berechnungen parallel ausgeführt werden, was die Ausführung der Anwendung beschleunigen kann.

Erweiterbarkeit

Bei der Erstellung konkreter constraintbasierter Anwendungen ist die Modellierung des Problems in der Constraintdomäne, die von dem zu verwendenden Constraintlöser unterstützt wird, oft sehr kompliziert und unintuitiv. Zudem entsprechen die Modelle oft nicht genau den Anforderungen der Anwendung, denn die verwendeten Constraints haben oft eine sehr viel allgemeinere Semantik, im konkreten Fall benötigt. Daher ist es oft geboten, spezielle Constraints und Variablendomänen für spezielle Anwendungen zu verwenden. So kann man einerseits das constraintbasierte Modell des konkreten Problems direkter aus der Problembeschreibung entnehmen und andererseits speziellere Constraints verwenden, die z.B. oft auftretende Fälle besonders effizient bearbeiten können.

Das Ausführungsmodell, das z.B. durch Propagation und Suche Lösungen für das Modell eines CSP sucht ist aber auch bei unterschiedlichen Constraints und Constraintdomänen immer gleich. Daher sollte ein Constraintlöser diese Funktionalität gekapselt zur Verfügung stellen. Ein Konzept, das diesen Weg besonders intensiv verfolgt, ist die Sprache CHR [Frü98], in der man Constraints individuell durch Regeln definieren kann, die dann in einem festen Ausführungsmodell als Constraintlöser funktionieren. Die neuen Constraints und Domänen sollten einfach in einen bestehenden Constraintlöser integrierbar sein, ohne dabei ineffizienter zu sein als vorhandene, abstraktere Modelle. Das bedeutet, daß man in eine Constraintlöser erstens neue Constraints mit individuell definierbarem Propagationsverhalten und zweitens neue Typen von Constraintdomänen integrieren können sollte.

Kapselung

Um durch die Erweiterung von Constraintlösern durch neue Constraints und Constraintdomänen keine großen und unübersichtlichen Systeme zu erhalten, sollten solche Erweiterungen in einem gegeben Rahmen als zusätzliche Pakete gehandhabt werden können. Es bietet sich an, einen Constraintlöser als generische Komponente umzusetzen, die mit unterschiedlichen Constraintdomänen parametrisiert werden kann. Der Constraintlöser sollte selbst ausschließlich abstrakte Operationen zur Verarbeitung beliebiger Constraints zur Verfügung stellen. Um auch heterogene Constraintprobleme, die mit unterschiedlichen Constraintdomänen beschrieben sind, bearbeiten zu können, sollte auch die Möglichkeit zur Kooperation zwischen den Paketen bestehen.

Suchalgorithmen sollten nach Möglichkeit gekapselte Komponenten und nicht fest im Solver integriert sein, was ich in Kapitel 5 ausführlicher begründen werde. Wenn zur Suche gekapselte Komponenten verwendet werden, kann man diese austauschen, um den besten Algorithmus für spezielle Problem explorativ zu finden. Dies wird z.B. in JCL [JCL] oder POOC [SR02] besonders gut ermöglicht, weil hier der Solver bzw. Suchalgorithmus während der Laufzeit ausgetauscht werden kann. Desweiteren erlaubt die Kapselung von Suchalgorithmen und damit von Suchräumen die Verwendung der Algorithmen in verteilten Anwendungen [Sch97, HvRBS98].

3.2 Basiskonzepte

Zusammenfassung. Die theoretischen Konzepte von Constraintlöser und Constraint werden in ihrer speziellen Umsetzung für *Asynchronous Constraint Solving* (ACS) beschrieben. ACS Constraintlöser bestehen im Wesentlichen aus einem Puffer, in dem anstehende Arbeitsaufgaben gespeichert werden, deren (verzögerte) Ausführung Constraintprobleme löst. ACS Constraints sind intensionale Beschreibungen von Constraints, zur effizienten Bearbeitung von Constraintproblemen in meinem Ausführungsmodell.

Beim Asynchronen Constraintlösen (ACS) wird der Constraintlöser als *blackbox* betrachtet, in der man Constraints absetzen und zurücknehmen kann. Diese Constraints sind für die jeweilige Anwendung durch Propagationsfunktionen (Def. 2.2.1) definiert. Die deklarative Semantik (Def. 2.1.3) eines Constraints wird dabei in den Propagationsfunktionen operational so umgesetzt, daß unzulässige Werte aus den Variablendomänen entfernt werden. Das Absetzen eines Constraints im Constraintlöser und die Ausführung der dazugehörigen Propagationsfunktionen wird in meinem Ansatz asynchron behandelt, d.h. die zweite Operation folgt nicht unbedingt unmittelbar auf die erste.

Durch die Trennung von Absetzen eines Constraints und der Ausführung von dessen Propagationsfunktionen kann in asynchronen Constraintlösern die Ausführung bestimmter Funktionen verzögert oder vorgezogen werden. Um ein solches *scheduling* über Propagationsfunktionen ausgehend von Expertenwissen über die Propagation der verwendeten Constraints zu unterstützen, wird jeder Propagationsfunktion beim ACS eine individuelle Wichtigkeit zugeordnet.

Definition 3.2.1 (Propagator) Ein Propagator $p = (f, w)$ ist gegeben durch eine Propagationsfunktion f und eine Wichtigkeit $w \in \{\bot, ..., \top\}$.

Asynchrone Constraintlöser erlauben außer dem inkrementellen Hinzufügen von Constraints auch das Zurücknehmen zuvor abgesetzter Constraints. So ergibt sich als eine weitere Aufgabe, die vom Solver erledigt werden muß, die Ausführung einer Constraint-Rücknahme. Diese wird genauso wie die Propagatoren asynchron ausgeführt und in einem internen Puffer solange gespeichert, bis ihre Ausführung angestoßen wird. Der interne Aufgabenpuffer eines Asynchronen Solvers speichert also alle noch anstehenden Aufgaben durch Referenzen auf Propagatoren oder Constraints, die zurückgenommen werden sollen.

Definition 3.2.2 (Aufgabenpuffer) Ein *Aufgabenpuffer* $A = \langle a_1, ..., a_n \rangle$ besteht aus einer Sequenz von Aufgaben a_i mit $a_i = post(c, p)$ oder $a_i = retract(c)$. Dabei ist c ein Constraint und $p \in \{\bot, ..., \top\}$ eine Wichtigkeit. Die Funktion insert fügt eine Menge von Propagatoren eines Constraints c in einen Aufgabenpuffer ein: $\mathsf{insert}(c, \{(i_1, w_1), ..., (i_n, w_n)\}, A) = A \dot\cup \{post(c, w_1), ..., post(c, w_n)\}$. Die Funktion $\mathsf{retract}$ fügt eine Referenz auf ein zu löschendes Constraint in einen Aufgabenpuffer ein: $\mathsf{retract}(c, A) = A \dot\cup \{retract(c)\}$.

Die Einfügeoperation $\dot\cup$ aus Def. 3.2.2 kann in unterschiedlichen Puffern unterschiedlich definiert werden. Die Aufgaben werden dann immer von der ersten Stelle des Puffers zur Ausführung entnommen (Abb. 3.3). Durch unterschiedliche Verwaltung der Puffer können sehr große Unterschiede bei der Performanz des Constraintlösers entstehen, die in Abschnitt 7.1 ausführlich untersucht werden. Normalerweise sollten die Prioritäten der Propagatoren eine wichtige Rolle bei der Verwaltung des Aufgabenpuffers spielen. Dies kann aber durch individuelle Implementierungen des Puffers beliebig festgelegt werden. Die Prioritäten der einzelnen Propagatoren verwendeter Constraints können ebenfalls individuell festgelegt werden, so daß sich hiermit eine präferierte Ausführungsreihenfolge der Propagationsfunktionen für jedes Problem definieren läßt. An dieser Stelle ist Expertenwissen über Constraintpropagation, wie z.B. *fairness* [HSW93], wichtig für das Finden der besten Abstimmung.

Wird ein CSP in einen Constraintlöser eingegeben, indem alle Constraints des CSP abgesetzt werden, ist a priori oft nicht klar, ob das CSP überhaupt lösbar ist. Sollte die Sequenz der abgesetzten Constraints inkonsistent sein, muß der Solver in irgendeiner Weise reagieren, denn die abgesetzten Constraints sollten wie ein *truth-maintenance-system* [Doy79] immer ein konsistentes CSP beschreiben. Insbesondere während der Ausführung nicht-deterministischer Suchalgorithmen in bereits durch Propagation eingeschränkten Suchräumen kann es durch falsche Instantiierungen oft zu solchen Situationen kommen. Weil aber die Reaktion auf das Auftreten von Inkonsistenzen stark vom Kontext (während der Suche, bei Benutzerinteraktion, während des Einlesens eines Constraintnetzes,...) abhängt, gebe ich keine allgemeine Vorgehensweise zur Reaktion an, es wird also nicht wie bei der CLP immer *backtracking* eingeleitet. Stattdessen wird, wie z.B. auch in [FFS98] jedem Solver eine Operation zur Verfügung gestellt, die Inkonsistenzen beseitigt. Während der Anwendung kann diese Operation dann für jeden konkreten Kontext angegeben werden. Bei der Implementierung von Suchalgorithmen zum Beispiel wird eine spezielle Reparaturoperation angegeben, die bestimmte Instantiierungen zurücknimmt und andere absetzt. In Kapitel 5 werden solche Rücknahmeoperationen angegeben.

Definition 3.2.3 (Reparaturoperation) Eine *Reparaturoperation* ρ ordnet jeder inkonsistenten Constraintsequenz C ein erfüllbare Constraintsequenz $\rho(C)$ zu, indem Constraints entfernt werden: $\varepsilon(\rho(C)) \subset \varepsilon(C)$

Die Existenz einer solchen Reparaturoperation ist in jedem Fall gesichert, weil man immer auch alle Constraints weglassen kann und so zur leeren Constraintsequenz (die ja konsistent ist) kommt.

Jeder asynchrone Constraintlöser verfügt über eine Referenz auf eine Reparaturoperation. Die zentrale Komponente des Solvers ist aber der Aufgabenpuffer, in dem Referenzen auf anstehende Aufgaben verwaltet werden. Weil die Aufgaben asynchron ausgeführt werden, ist es für eine Constraint-basierte Anwendung normalerweise nicht klar, inwieweit die dazugehörigen Propagations- bzw. Rücknahmeoperationen schon ausgeführt wurden. Daher stellt ein asynchroner Constraintlöser immer eine Information über den aktuellen Stand der Berechnungen zur Verfügung. Dieser Stand wird durch einen Zuverlässigkeitswert angegeben, der mit fortschreitender Abarbeitung von Aufgaben steigt und wenn neue Aufgaben gepuffert werden, sinkt.

Definition 3.2.4 (Asynchroner Constraintlöser) Eine *Asynchroner Constraintlöser* $acs = (z, A, \rho)$ ist gegeben durch

- einen Zuverlässigkeitswert $z \in \{inconsistent, \bot, ..., \top\}$,
- einen Aufgabenpuffer A und

- einer Reparaturoperation ρ.

Bemerkung 3.2.1
Der Zuverlässigkeitswert eines asynchronen Constraintlösers kann unterschiedlich definiert werden. Er steht aber immer in Beziehung zu den noch anstehenden Aufgaben, so daß er auch aus dem Puffer direkt berechnet werden kann. Ich deklariere daher die Abbildung zuv, *die jedem Aufgabenpuffer einen Zuverlässigkeitswert zuordnet, für die gilt:* $\mathsf{zuv}(\langle\rangle) = \top$. *In der Regel berechnet sich der Wert aus den Wichtigkeiten der eingefügten Propagatoren. Ist der Zuverlässigkeitswert* $z = inconsistent$, *so bedeutet das, daß eine Inkonsistenz aufgetreten ist, die gerade beseitigt wird. In diesem Fall kann eine Variablendomäne auch unzulässige Werte enthalten, was sonst nur dann möglich ist, wenn Constraints zurückgenommen wurden und die Rücknahmeoperation noch nicht (vollständig) ausgeführt wurde.*

In der aktuellen ACS Implementierung J.CP kann der Zuverlässigkeitswert eines Solvers nur die Werte $inconsistent, \top$ *oder* $\perp$ *annehmen. Dabei steht* $\perp$ *dafür, daß noch Aufgaben anstehen, die anderen Werte wurden bereits beschrieben. Die Implementierung der Funktion* zuv *sollte möglichst die Eigenschaft haben, daß sie monoton fällt, wenn keine weiteren Constraints abgesetzt werden. Dies ist eine nicht-triviale Aufgabe, weil bei der Propagation immer wieder alte Constraints (mit wichtigen Aufgaben) neu in den Puffer des Solvers eingefügt werden können.*

Die extensionale Beschreibung von Constraints wie in Def. 2.1.3 ist nicht zur Implementierung geeignet, weil die Ausführung von Propagation über Tupelmengen zu ineffizient ist (z.B. im Konfigurationswerkzeug [See01]). Wie in allen effizienten Constraintlösern (z.B. [CD96, JMSY92]) definiere ich daher eine intensionale Beschreibung von Constraints zur Implementierung. Jedes Constraint stellt einen oder mehrere Propagatoren zur Verfügung, die zur Konsistenzprüfung und zur Propagation bei der Ausführung von Constraintprogrammen dienen. Enthalten die aktuellen Domänen der Variablen keine Werte mehr, die in der extensionalen Beschreibung des Constraints enthalten sind, so erzeugt die Propagationsfunktion eine Inkonsistenz. Ansonsten werden die Domänen so eingeschränkt, daß möglichst viele Werte aus den Domänen entfernt werden, die nach aktuellem Wissen nicht mehr in einer Lösung enthalten sein können.

Außer Referenzen auf die Variablen des Constraints und den Solver, in dem es abgesetzt wurde, wird auch der aktuelle Zustand des Constraints gespeichert. Dieser Zustand wird durch die Operationen "Absetzen" und "Zurücknehmen" bestimmt, ein Constraint gilt als aktiv, wenn es abgesetzt und nicht zurückgenommen wurde.

Weil ACS für die nebenläufige Anwendung konzipiert ist, sollen keine globalen Datenstrukturen im Ausführungsmodell verwendet werden. Insbesondere globale Constraintspeicher oder eine globale Verwaltung ehemaliger Zustände soll verhindert werden. Zustände werden in vielen Constraintverarbeitenden Systemen gespeichert, um falsche Entscheidungen rückgängig machen zu können. Bei der nichtdeterministischen Wertauswahl während der Suche nach Lösungen für NP-vollständige Probleme zum Beispiel, werden oft falsche Entscheidungen getroffen. In ACS wird dieses Problem nicht durch Wiederherstellen ehemaliger Zustände, sondern durch die Operationen zum Absetzen und Zurücknehmen von Instantiierungen(also Constraints) gelöst. Diese beiden Operationen sind die einzigen Möglichkeiten zur Eingabe von Wissen in den Constraintlöser und sind desweiteren gegenseitg Umkehroperationen. Damit kann jede (falsche) Eingabe durch den Aufruf ihrer Umkehroperation rückgängig gemacht werden. Um diese Operationen zu ermöglichen werden einerseits Abhängigkeiten zwischen Constraints, die durch Propagation entstehen und andererseits ehemalige Variablendomänen lokal in den Constraints gespeichert. Aus gespeicherten Variabledomänen (Domänenurbild) werden bei der Constraint-Rücknahme erweiterte Variablendomänen berechnet. Aus den gespeicherten Abhängigkeiten (Konsequenzen) wird dann der gesuchte globale Zustand hergestellt.

Definition 3.2.5 (ACS Constraint)
Ein *ACS Constraint* $c = (\overline{v}, T, acs, z, E, \overline{h})$ ist gegeben durch

- einen Variablenvektor $\overline{v} = (v_1, ..., v_n)$,
- eine Menge von Propagatoren $T = \{(f_1, w_1), ..., (f_m, w_m)\}$ mit $i \neq j \Rightarrow w_i \neq w_j$, für $1 \leq i, j \leq m$,
- einen asynchronen Constraintlöser acs,
- einen Zustand $z \in \{active, inactive\}$
- eine Menge von Constraints E (Konsequenzen von c genannt) und
- einen Vektor $\overline{h} = (\overline{p}_1, ..., \overline{p}_n)$ von Domänenbeschreibungen mit $\mathsf{dom}(v_i) \subseteq \mathsf{dom}(\overline{p}_i)$ (Domänenurbild von c genannt).

Durch die Propagatoren eines Constraints können laut Def. 3.2.1 und 2.2.1 Variablendomänen eingeschränkt werden. Die Werte, die durch die einmalige Ausführung einer Propagationsfunktion eines Constraints c aus den Domänen aller Variablen von c entfernt werden, nenne ich Reduktion der Propagationsfunktion.

Definition 3.2.6 (Reduktion) Sei eine ACS Propagationsfunktion f_i eines Constraints $c = (\overline{v}, \{(f_1, w_1)..., (f_n, w_n)\}, acs, z, E, \overline{h})$ gegeben, dann ist die *Reduktion* von f_i definiert durch $\mathsf{red}(f_i)(\overline{v}) := \bigcup \varepsilon(\mathsf{dom}^+(\overline{v})) \setminus \bigcup \varepsilon(\mathsf{dom}^+(f_i(\overline{v})))$. Dabei ist $\mathsf{dom}^+((v_1, ..., v_n)) := (\mathsf{dom}(v_1), ..., \mathsf{dom}(v_n))$.

Die Reduktion der Propagatoren eines Constraints ist laut Def. 2.2.1 abhängig von den aktuellen Variablendomänen vor der Ausführung der Propagationsfunktionen. Setzt man diese als gegeben voraus, so kann durch die Reduktion für jede Variable eine bestimmte Domäne berechnet werden, die nach dem aktuellen Wissensstand keine Werte enthält, die garantiert nicht Bestandteil einer Lösung sind. Durch Bildung des Kreuzprodukts über den so eingeschränkten Domänen erhält man also eine Menge von Vektoren, die alle Lösungen des Constraints enthält. Daraus ergibt sich die Tatsache, daß ein ACS Constraint durch seine Propagatoren eine Menge von Variablenbelegungen bestimmt und somit ist Def. 2.1.3 auch für ACS Constraints anwendbar.

Lemma 3.2.1 Jedes ACS Constraint ist ein Constraint.

Beweis:
Sei ein beliebiges ACS Constraint $c = (\overline{v}, T, acs, z, H, \overline{h})$ gegeben, so daß $\mathsf{vars}(c) =$

$\overline{v} = (v_1, ..., v_n)$ gilt.

Ich wähle für jedes $v_i \in \varepsilon(\overline{v})$ die Constraintdomäne $\mathsf{dom}(v_i)$, dann gilt wegen Def.2.2.1 für $T = \{(f_1, w_1), ..., (f_n, w_n)\}$:

$$\forall (f,w) \in T.f(\times(\mathsf{dom}^+(\overline{v}))) \subseteq \times(\mathsf{dom}^+(\overline{v}))$$

$$\overset{Def.\text{ von }\mathsf{dom}^+\text{ in }3.2.6}{\Rightarrow}$$

$$\forall (f,w) \in T.f(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n)) \subseteq (\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n))$$

Damit ist $\bigcap_{(f,w)\in T} f(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n)) = \mathsf{sem}(c)$ die deklarative Semantik von c.

□

3.3 Programmierkonzepte und deren Realisierung

Zusammenfassung. Zur Implementierung Constraint-basierter Anwendungen wird eine Erweiterung imperativer Programmiersprachen um Anweisungen zur Constraintverarbeitung angegeben. Constraints definieren Propagationsmethoden, die bei der Ausführung des Constraintprogramms aufgerufen werden und zu Domänenreduktionen führen. Die Formalisierung dieser Domänenreduktion und die Definition weiterer für die Constraintverarbeitung relevanter Eigenschaften von ACS Constraints und Constraintprogrammen wird in diesem Abschnitt angegeben. Es werden Zusammenhänge zwischen Domänenreduktionsfunktionen im Sinne von [Apt98] und der operationalen Semantik von ACS Constraints aufgezeigt.

Bei der Erstellung eines constraintbasierten Anwendungsprogramms müssen in Analyse und Entwurf der Software zusätzliche Entscheidungen zur Modellierung des Problems mit Constraints gefällt werden. Oft wird ein bestimmter Constraintlöser verwendet, der Probleme weniger bestimmter Constraintdomänen lösen kann. Das Problem der "realen Welt" muß dann im jeweiligen Eingabeformat des gewählten Constraintsystems, also z.B. prädikatenlogische Formeln in CLP, ausgedrückt werden (z.B. [GM00]). Das generische Ausführungsmodell ACS ist konzeptionell an keine bestimmte Constraintdomäne gebunden, so daß ich in diesem Kapitel davon ausgehe, daß die Constraints, die man zur constraintbasierten Modellierung eines gegebenen Problems braucht, als implementierte ACS Constraints vorliegen. In meiner ACS-Implementierung J.CP [Rin02b], sind die bekannten Constraintsysteme "endliche Mengen" und "Mengen ganzer Zahlen" als gekapselte Constraintsysteme gegeben, die evtl. mit neuen anwendungsbezogenen Constraints kombiniert in Anwendungsprogrammen verwendet werden können (z.B. in $\mathcal{M}$Plan).

Sind also alle Constraints, die zur Modellierung des Anwendungsproblems verwendet werden sollen, gegeben, so können diese in einem Constraintprogramm (der Anwendung) abgesetzt werden. Dies bedeutet in ACS, wie in jedem anderen inkrementellem Constraintlöser, daß die Domänen der Variablen des abgesetzten Constraints möglichst nur noch Werte enthalten, die in einer Lösung des durch alle abgesetzten Constraints definierten CSP enthalten sind. Wie weit solche Domäneneinschränkungen gemacht werden können, hängt von der Implementierung der Propagationsfunktionen bzgl. einer angestrebten Konsistenz (Def. 2.2.2) ab. Im Zusammenhang mit den Domäneneinschränkungen prüft jedes Constraint bei seiner Ausführung auch seine Konsistenz. Kann ein Constraint die Domäne einer Variablen so weit einschränken, daß sie keine Werte mehr enthält, so ist eine Inkonsistenz aufgetreten, weil die Variable nicht mehr instantiiert werden kann. In ACS wird dies im Constraintlöser signalisiert, indem sein Zuverlässigkeitswert auf *inconsistent* gesetzt wird. In den meisten anderen Systemen wird automatisch ein Suchalgorithmus angestoßen, der versucht, eine zulässige Alternative im Suchbaum zu finden. Dieses Vorgehen ist nicht in jedem Fall geeignet wie ich es bei der Einführung des Begriffs der Reparaturoperation (Def. 3.2.3) bereits beschrieben habe. Deshalb lasse ich dem Programmierer der Anwendung die Möglichkeit, kontextbezogen auf die Inkonsistenz zu reagieren, indem er eine individuelle Reparaturoperation angibt.

Constraintprogramme sind in ACS als imperative Programme definiert, was die Implementierung des Ausführungsmodells und der Anwendungen in imperativen und objektorientierten Sprachen unterstützt. In Sprachen wie z.B. Java sind Konzepte und APIs zur Implementierung von Nebenläufigkeit und Verteilung bereits gegeben, so daß sie die Implementierung von ACS direkt unterstützen. Funktionale Programmiersprachen sind meiner Meinung nach nicht zur CP geeignet, weil die Verarbeitung der Constraints im Hintergrund dort stets als Seiteneffekt implementiert werden müßte, was der funktionalen Philosophie widerspricht. Die klassische Programmierumgebung der CP, nämlich der logischen Programmierung, habe ich nicht gewählt, weil dort das Konzept der Nebenläufigkeit nicht gut umzusetzen ist [SR00, HvRBS98]. Hierzu wurde in der Vergangenheit zwar viel Forschungsarbeit geleistet (z.B. [Sar93]), es hat sich aber immer herausgestellt, das die systeminhärente Suche von Prolog nur schwer verteilt zu implementieren ist.

Die wesentlichen Operationen, die ein *black-box*-Constraintlöser, wie ich ihn motiviert habe, zur Verfügung stellen sollte, sind das Hinzufügen und das Löschen von Wissen als Eingaben, sowie das Lesen inferierten Wissens als Ausgabe. Bei der Erstellung eines Constraintprogramms werden also Operationen zum Absetzen und Löschen von Constraints sowie zum Lesen von Variablendomänen verwendet. Die Definition der einzelnen Komponenten entspricht der in der OOP üblichen Erzeugung der Objekte bzw. der in der imperativen Programmierung üblichen Deklaration und Definition von Variablen. Wegen der nebenläufigen Ausführung der Constraintverarbeitung ist es in den meisten Anwendungen notwendig, vor dem Lesen von Variablendomänen den Stand der aktuellen Propagation zu kennen, so daß man den Zuverlässigkeitswert der Constraintlöser als zusätzliche Ausgabe der *black-box* verwenden kann.

Definition 3.3.1 (Constraintprogramm) Ein *Constraintprogramm* ist ein imperatives Programm mit folgenden zusätzlichen Anweisungen

1. Definition von Constraintlösern: `acs = new Solver(`ρ`)`, wobei ρ die (vorläufig zu verwendende) Reparaturoperation von `acs` ist.

2. Definition von Constraintvariablen: `v = new Variable(`$\overline{p}$`)` mit der initialen Domäne $\mathsf{dom}(\overline{p})$

3. Definition von Constraints: `c = new Constraint(`$\overline{v}$`,`T`)`, wobei $\overline{v}$ ein Variablenvektor und T eine endliche Menge von Propagatoren ist
4. Constraint absetzen: `c.post(acs)` in einem definierten Constraintlöser `acs`.
5. Constraint zurücknehmen: `c.retract()`
6. Domänenanfragen $\overline{p}$ `=` v`.getDomain()` für definierte Constraintvariablen v
7. Zuverlässigkeitsanfragen `x = acs.getReliability()` in einem definierten Constraintlöser `acs`.

Zur einfacheren Handhabung der einzelnen Eigenschaften und definierten Komponenten von Constraintprogrammen im weiteren Verlauf dieser Arbeit, definiere ich einige Operationen zur Analyse von Constraintprogrammen.

Definition 3.3.2 (Komponenten eines Constraintprogramms) Sei ein endliches Constraintprogramm P gegeben, dann sind alle in P definierten Constraintlöser in der Menge $\mathsf{solvers}(P)$ und alle definierten Constraintvariablen in $\mathsf{vars}(P)$ enthalten. Die Menge der definierten und abgesetzten und nicht zurückgenommenen Constraints wird $\mathsf{constraints}(P)$ genannt. Die Menge aller Propagatoren ist definiert durch $\mathsf{propagators}(P) := \bigcup T$, mit $(\overline{v}, T, acs, z, E, \overline{h}) \in \mathsf{constraints}(P)$. Die Menge aller in P vorkommenden Anweisungen ist definiert durch $\mathsf{instr}(P)$.

Bemerkung 3.3.1
In einem kontinuierlichen, also unendlichen Constraintprogramm sind die angegebenen Mengen aus Def. 3.3.2 zu jedem Zeitpunkt nach Programmstart ebenfalls endlich.

Bemerkung 3.3.2
Die Menge $\mathsf{constraints}(P)$ *enthält nur diejenigen Constraints, die nach Def. 3.3.1.3 definiert wurden und danach wie in Def. 3.3.1.4 abgesetzt wurden und nicht danach durch eine Anweisung wie in Def. 3.3.1.5 zurückgenommen wurden. Dasselbe Constraint kann innerhalb eines Programms auch mehrfach abgesetzt und zurückgenommen werden, dabei ist für die Menge* $\mathsf{constraints}(P)$ *entscheidend, welches die letzte vorkommende Operation war. Aus der Menge* $\mathsf{constraints}(P)$ *und den Variablen von* P *kann ein CSP im Sinne von Def. 2.1.10 durch Ordnung der Constraints abgeleitet werden.*

Eine weitere technisch besonders relevante Eigenschaft von Constraintprogrammen ist, ob Constraints zurückgenommen werden oder nicht. Das Zurücknehmen von Constraints bzw. das Wiederherstellen ehemaliger Zustände ist in den meisten CP-Anwendungen eine erforderliche Operation, weil die dort implementierten Probleme in der Regel nicht deterministisch gelöst werden können. Aus technischen Gründen definiere ich die Additivität von Constraintprogrammen, um später zunächst die Korrektheit meines Ausführungsmodells ohne Verwendung von Constraint-Rücknahme zeigen zu können.

Definition 3.3.3 (additiv) Ein Constraintprogramm, in dem keine Constraints zurückgenommen (Def. 3.3.1.5) werden, heißt *additiv*.

Die operationale Semantik der Programmanweisungen aus Definition 3.3.1 wird durch die Änderungen des Systemzustands beschrieben, wie es bei der Definition der Semantik imperativer Sprachen (z.B. [Hoa69, OH83]) üblich ist. Der Teil des Systemzustands, der für die Constraintverarbeitung relevant ist, ist durch die verwendeten ACS Komponenten gegeben, deren Veränderung aufgrund von Programmanweisungen in der folgenden Definition angegeben werden. Dabei wird der Wert einer Variable v zum Zeitpunkt t mit v^t beschrieben, während ihr Wert nach der Ausführung bestimmter Operationen durch v^{t+1} angegeben wird.

Definition 3.3.4 (Ausführung von Constraintprogrammen) Die Anweisungen in einem Constraintprogramm werden wie folgt umgesetzt:

1. `acs = new Solver(`ρ`):`
   acs$= (\top, \langle\rangle, \rho)$

2. `v = new Variable(`$\overline{p}$`):`
   v$= (\overline{p}, \{\emptyset_{P_1}, ..., \emptyset_{P_n}\})$

3. `c = new Constraint(`$\overline{v}, T$`):`
   c$= (\overline{v}, T, null, inactive, \emptyset, null)$

4. `c.post(s)` mit $\mathtt{s}^t = (z, A, \rho)$ und $\mathtt{c}^t = (\overline{v}, T, x, y, E, \overline{h})$ und $x \in \{null, \mathtt{s}\}$:
   $\mathtt{c}^{t+1} = (\overline{v}, T, s, active, E, \overline{h})$
   $\wedge \mathtt{s}^{t+1} = (\mathsf{zuv}(A'), A', \rho)$,
   wobei: $A' = \mathsf{insert}(\mathsf{c}, T, A)$

5. `c.retract()` mit $s^t = (z, A, \rho)$ und $\mathtt{c}\ ^t = (\overline{v}, T, s, x, E, \overline{h})$:
   $\mathtt{c}^{t+1} = (\overline{v}, T, s, inactive, E, \overline{h})$
   $\wedge\ s^{t+1} = (z, A', \rho)$,
   wobei: $A' = \mathsf{retract}(\mathsf{c}, A)$

6. $\overline{p}$ `= v.getDomain()`, mit v$= (\overline{p}', C)$:
   $\overline{p} = \overline{p}'$

7. `x = acs.getReliability()`, mit acs$= (z, A, \rho)$:
   x$= z$

Bemerkung 3.3.3
Offensichtlich besteht in einem additiven Constraintprogramm die einzige Möglichkeit die Domäne einer Variablen v zu verändern darin, ein Constraint über v abzusetzen.

Fallstudie, Teil 5
Im Terminplaner $\mathcal{M}$Plan wird ein Terminvorschalg `prop` *interaktiv durch die Methode* `newApp()` *eingelesen. Die Eingaben werden als Constraintvariablen interpretiert und es werden Constraints abgesetzt, die die Überlappung von Terminen verhindern (vgl. disjoint-Constraint in Fallstudie Teil 3) und den Termin anzeigen sobald er feststeht (vgl. "zeigt an" in Fallstudie Teil 2). Im folgenden ist der Java-Code, der das Vorschlagen von Terminen in $\mathcal{M}$Plan implementiert dargestellt.*

```
class MPlan{
...
  //Solver global deklarieren und starten
  Solver acs = new Solver(r);

  void newApp(){
     Proposal prop = readProposal(); // Vorschlag einlesen

     // Constraintvariablen für Werte erzeugen und initialisieren
     Variable tag = new TagVar(prop.getTag());
     Variable zeit = new ZeitVar(prop.getZeit());
     Variable dauer = new FDVar(prop.getDauer());
     Variable ort = new EnumVar(prop.getOrt());

     // Appointment-Objekt Erzeugen
     Appointment app = new Appointment(tag,zeit,dauer,ort);

     // Falls der Tag bereits festgelegt wurde
     if(tag.instantiated()){
         // alle bekannten Termine des Tages
         Enumeration apps = database.getApps(tag);
         while(apps.hasMoreElements()){
             //Überschneidungsfreiheit sichern
             Constraint n = new Disjoint(app,apps.nextElement());
            n.post(acs)
         }
     }

     // ZeigtAn aktualisiert GUI,
     // sobald alle Variablen aus app instantiiert sind
     Constraint zeige = new ZeigtAn(app);
     zeige.post(acs);
     ...
}
...
}
```

Bei der Ausführung eines Constraintprogramms werden in ACS also die Propagatoren der Constraints (Def. 3.3.4.4) bzw. Referenzen auf zurückgenommene Constraints (3.3.4.5) in definierten Constraintlösern gepuffert. Die Ausführung der Propagationsfunktionen dieser Propagatoren werden dann in einem nebenläufigen Prozess angestoßen. Die Funktionen selbst sind in ACS durch imperative Programme bzw. Methoden definiert. Um als Propagationsfunktion gemäß Def. 2.2.1 verwendet werden zu können, muß eine ACS Propagationsmethode Variablendomänen einschränken können. Zusätzlich ist in ACS im Unterschied zu den meisten anderen bekannten Ausführungsmodellen auch erlaubt, daß in der Propagationsmethode eines Constraints selbst auch wieder Constraints erzeugt und abgesetzt oder zurückgenommen werden. Durch diese Möglichkeit kann es zu nicht-terminierenden Ketten von Constraintaufrufen kommen, die die Terminierung der Propagation verhindern. Solche unendlichen

Ketten sind bei der Implementierung von ACS Constraints durch Propagationsmethoden zu verhindern. Welche Zusammenhänge von verschachtelten Constraintaufrufen dabei erlaubt bzw. unzulässig sind, werde ich in Abschnitt 3.4 beschreiben. Ich erlaube trotz der genannten (theoretischen) Schwierigkeiten das verschachtelte Absetzen von Constraints, um bestimmte komplexe Constraints, wie z.B. *labelling*-Constraints (Abschnitt 5), als Komposition einfacherer Constraints implementieren zu können.

Definition 3.3.5 (ACS Propagationsmethode) Eine *ACS Propagationsmethode* ist ein imperatives Programm mit folgenden zusätzlichen Anweisungen

1. Definition von Constraints: `c = new Constraint(`$\overline{v}, T$`)` wobei $\overline{v}$ ein Variablenvektor und T eine Menge von Propagatoren ist
2. Constraint absetzen: `c.post(s)`, wobei `s` der Asynchrone Constraintlöser ist, in dem die Propagationsmethode ausgeführt wird.
3. Constraint zurücknehmen: `c.retact()`
4. Erzeugen von Inkonsistenz-Ereignissen:
   `broadcast(new InconsistencyEvent())`
5. Domänenanfragen $\overline{p}$ `= v.getDomain()` für definierte Constraintvariablen `v`
6. Definition von Variablendomänen: `Entail = v.setDomain(`$\overline{p}$`)`

Ebenso wie die operationale Semantik von Constraintprogrammen definiere ich nun auch die Semantik von Propagationsmethoden durch Zustandsübergänge in den ACS Komponenten.

Definition 3.3.6 (Ausführung von Propagationsmethoden) Sei ein ACS Constraint $c = c^t = (\overline{v}, T, acs, z, E, \overline{h})$ gegeben, wobei die Propagationsfunktionen der Propagatoren T durch ACS Propagationsmethoden implementiert sind. Dann werden die Anweisungen der ACS Propagationsmethoden wie folgt umgesetzt:

1. d `= new Constraint(`$\overline{v}', T'$`)`, mit $acs = (z, A, \rho)$:
   $d = (\overline{v}', T', acs, inactive, \emptyset, null)$
2. d`.post(`acs`)`, mit $acs^t = (z, A, \rho)$ und $d^t = (\overline{v}', T', acs, x, E', \overline{h}')$:
   $d^{t+1} = (\overline{v}', T', acs, active, E', \overline{h}')$
   $\wedge\, acs^{t+1} = (\mathsf{zuv}(A'), A', \rho)$,
   wobei gilt: $A' = \mathsf{insert}(d, T', A)$
3. d`.retact()`, mit $acs^t = (z, A, \rho)$ und $d^t = (\overline{v}', T', acs, x, E', \overline{h}')$:
   $d^{t+1} = (\overline{v}', T', acs, inactive, E', \overline{h}')$
   $\wedge\, acs^{t+1} = (\mathsf{zuv}(A'), A', \rho)$,
   wobei gilt: $A' = \mathsf{retract}(d, A)$
4. `broadcast(new inconsistencyEvent())`, mit $acs^t = (z, A, \rho)$:
   $acs^{t+1} = (inconsistent, A, \rho)$
5. $\overline{p}$ `= v.getDomain()`, mit `v`$\in \varepsilon(\overline{v})$ und `v` $= (\overline{p}', C)$:
   $\overline{p} = \overline{p}'$

6. `Entail = v.setDomain(`$\overline{p}$`)`, mit $\mathtt{v}^t = (\overline{p}', (C)_{P\in\{P_1,...,P_n\}}) \in \varepsilon(\overline{v})$:
$c^{t+1} = (\overline{v}, T, acs, z, E \cup \{\mathtt{Entail}\}, \overline{h})$
$\wedge \mathtt{v}^{t+1} = (\langle p''_1, ..., p''_n\rangle, (C')_{P\in\{P_1,...,P_n\}})$
$\wedge\, acs^{t+1} = (\mathsf{zuv}(A'), A', \rho)$,
wobei für jedes Merkmal P_i mit den Instanzen $p_i \in \varepsilon(\overline{p}), p'_i \in \varepsilon(\overline{p}')$ gilt:
$\mathsf{dom}(\langle p'_1, ..., p'_i, ..., p'_n\rangle) \supset \mathsf{dom}(\langle p'_1, ..., p'_{i-1}, p_i, p'_{i+1}, ..., p'_n\rangle) \Rightarrow$
$p''_i = p_i \wedge \mathtt{Entail} \supseteq C_{P_i} \wedge C'_{P_i} = C_{P_i} \circ c$
$\wedge\, \forall c' = (\overline{v}', T', (A, z, \rho), active, E', \overline{h}') \in C_{P_i} : A' = \mathsf{insert}(c', T', A)$
$\wedge\, \mathsf{dom}(\langle p'_1, ..., p'_i, ..., p'_n\rangle) \subseteq \mathsf{dom}(\langle p'_1, ..., p'_{i-1}, p_i, p'_{i+1}, ..., p'_n\rangle) \Rightarrow$
$p''_i = p'_i \wedge C'_{P_i} = C_{P_i}$

Zur Verdeutlichung meiner Umsetzung von Domänendefinitionen (Def. 3.3.6.6) gebe ich eine alternative Beschreibung in Form eines Programmfragments in Abb. 3.2 an.

$Entail := new\ ConstraintSet();$ //nochmals abgesetzte Constraints
foreach $p'_i \in \varepsilon(\overline{p}')$ **do** //Jedes Merkmal der Domäne untersuchen
  if $\mathsf{dom}(p_1, ..., p'_i, ..., p_n) \subset \mathsf{dom}(p_1, ..., p_i, ..., p_n)$
    then $p_i := p'_i$; //Domäne reduzieren
      foreach $c' = (\overline{v}, T, s, active, E, \overline{h}) \in C_{P_i}$ //aktive Constraints
        if $c \neq c'$ **then** $c'.post(acs)$ **fi**; //Propagation
        $Entail := Entail \cup \{c'\}$ //Konsequenz merken
      end
      if $c \notin C_{P_i}$ //bei erster Ausführung...
        then $C_{P_i} := \langle C_{P_i}, c\rangle$ //c als abhängig von C_{P_i} eintragen
      fi
  fi
end
return $Entail$;

Abbildung 3.2: Ausführung der Variablendefinition $Entail$ = v.`setDomain`($\overline{p}'$) in der Propagationsmethode eines Constraints c für $v = (\langle p_1, ..., p_n\rangle, (C_{P_1}, ..., C_{P_n}))$ gemäß Def. 3.3.6.6.

Fallstudie, Teil 6

Das `Disjoint`*-Constraint aus Fallstudie Teil 5 kann in der Sprache Java folgendermaßen als ACS Constraint für den Solver J.CP implementiert werden.*

```
class Disjoint extends Constraint{
   Disjoint(Appointment a, Appointment b){  //Konstruktor
      super();  // Allgemeinen Constraint-Konstruktor ausfuehren
      vars[0] = a; vars[0] = b;
      //Propagationsmethoden mit Priorität 5 absetzen
      acs.push(this,5,a,b);
   }

   // Die Variablen von this
   private Variable[] vars = new Variable[2];
```

```
   //Dispatcher fuer Propagatinsmethoden
   propagate(int x, Appointment a,b){
      if(x == 5) runDisjoint(a,b);
  }

  private runDisjoint(Appointment a, Appointment b){
     // gleicher Tag bereits instantiiert
     if(a.getTag().equals(b.getTag()){ // gleicher Tag
        // falls Verabredung a vor b sein muss...
        if(a.getZeit().getMax() + a.getDauer().getMax()
           < b.getZeit().getMin()+b.getDauer.getMin())
           // ...entsprechendes Constraint absetzen
           new Vor(a,b).post(acs);
        // falls Verabredung a nach b sein muss...
        if(b.getZeit().getMax() + b.getDauer().getMax()
           < a.getZeit().getMin()+a.getDauer.getMin())
           // ...entsprechendes Constraint absetzen
           new Vor(b,a).post(acs);

     }
     ... // weitere Propagationsalgorithmen
  }
}
```

Das Constraint `Vor` *wird immer dann ausgeführt, wenn sich aus den aktuellen Domänen der Variablen* `zeit` *und* `dauer` *der Appointment-Objekte eine Reihenfolge ableiten lässt. Dieses schränkt dann Domänen ein oder erzeugt Inkonsistenzen. Zur Berechnung von Reihenfolgen wird hier eine Abbildung in die ganzen Zahlen mit den Funktionen* `getMin` *und* `getMax` *vorgenommen.*

```
class Vor(){
...
  private runVor(a,b){
     // wenn fruehestes Ende von a nach spaetestem Beginn von b
     if(a.getZeit().getMin() + a.getDauer().getMin()
        > b.getZeit().getMax()){
         // Inkonsistenz verarbeiten
         broadcast(new InconsistecyEvent(this));
         return; // Methodenausführung abbrechen
     }

     // Ansonsten wird spaeteste Anfangszeit von a berechnet
     a.getZeit().setMaxZeit(minusZeit(b.getZeit().getMaxZeit(),
                                      a.getDauer().getMinZeit());
     // Propagation wird von a automatisch veranlasst, wenn noetig
  }
}
```

Die Einschränkung der Domäne von `a.zeit` *wird in der Methode* `runVor` *durch* `setMaxZeit` *durchgeführt, die den spätesten Startzeitpunkt von Termin* `a` *aus der Dauer von* `a` *und dem spätesten Zeitpunkt von Termin* `b` *berechnet. Damit ist das* `Vor`*-Constraint von* `a.dauer` *und* `b.zeit` *abhängig und wird immer* *dann erneut ausgeführt, wenn sich die Domäne einer dieser Constraintvariablen ändert. Für das Rechnen mit Zeiten werden spezielle Methoden wie z.B.* `minusZeit` *verwendet.*

Die Constraints, die in einer Propagationsmethode abgesetzt werden, können rekursiv durch die Constraints akkumuliert werden. Ich definiere eine Abbildung die jedem Constraint alle Constraints zuordnet, die direkt oder indirekt von ihm abgesetzt wurden, um diesen Zusammenhang bei den Voraussetzungen für die Terminierung der Propagation berücksichtigen zu können.

Definition 3.3.7 (Hilfsconstraint) Sei C die Menge der in einer ACS Propagationsmethode m erzeugten und abgesetzten Constraints (Def. 3.3.5.1,.2), dann wird die Menge der *Hilfsconstraints* von m, $\mathsf{builtin}(m)$ wie folgt definiert:

- $C \subseteq \mathsf{builtin}(m)$
- $\forall c' \in C.\mathsf{builtin}(c') \subseteq \mathsf{builtin}(m)$

Wobei die Menge $\mathsf{builtin}(c') := \bigcup_{m' \in \mathsf{propagators}(c')} \mathsf{builtin}(m')$ die Hilfsconstraints aller Propagationsmethoden von c' enthält.

Bemerkung 3.3.4
Die Menge der Hilfsconstraints eines Constraints c kann c selbst nie enthalten, weil es gemäß Def. 3.3.5 keine Möglichkeit gibt, eine Referenz auf ein Constraint als Parameter an Hilfsconstraints weiterzugeben.

Definition 3.3.8 (constraintfrei) Ein Constraint bzw. eine ACS Propagationsmethode c mit $\mathsf{builtin}(c) = \emptyset$ heißt *constraintfrei.*

Bemerkung 3.3.5
Wenn die Menge $\mathsf{builtin}(c)$ *endlich ist, gilt automatisch, daß keine endlosen Ketten von Hilfsconstraints aufgebaut werden, d.h. jedes Constraint benutzt "im Endeffekt" ausschließlich constraintfreie Hilfsconstraints.*

Bei ACS sollen bei der Ausführung von Propagationsmethoden neben den Domäneneinschränkungen auch Konsistenzprüfungen durchgeführt werden. Geht man bei der Umsetzung des Constraintbegriffs aus Def. 2.1.3 durch ACS Constraints (Def. 3.2.5) aus, so kann das Auftreten einer Inkonsistenz in den Propagationsmethoden festgestellt werden, indem überprüft wird, ob eine aktuelle Variablendomäne noch Werte enthält, die zu einer Lösung gehören können. Wann solche Konsistenzprüfungen durchgeführt werden, ist dem Programmierer der Propagationsmethode überlassen (Mehr dazu in Abschnitt 4.1). In meiner ACS Implementierung J.CP wird zusätzlich bei der Einschränkung von Variablendomänen immer ein Konsistenztest durchgeführt, der überprüft, ob die aktuelle Domäne noch mindestens einen Wert enthält. Ist dies nicht der Fall, so kann in J.CP auch von Variablendomänen ein entsprechendes Ereignis ausgelöst werden. Im Ausführungsmodell selbst ist dies aber nicht vorgesehen, so daß die Konsistenz eines Constraintprogramms (a posteriori) dadurch festgestellt werden kann, daß keine Inkonsistenz-Ereignisse aufgetreten sind.

Definition 3.3.9 (konsistentes Constraintprogramm) Ein Constraintprogramm heißt *konsistent,* wenn bei seiner Ausführung keine Inkonsistenz-Ereignisse (Def. 3.3.5.4) erzeugt werden.

Durch die Domäneneinschränkungen in den Propagationsmethoden (Def. 3.3.5.6) und den aktuellen Domänen der Variablen vor der Ausführung der Propagationsmethode, werden die verbleibenden Werte in den aktuellen Variablendomänen definiert. Das Kreuzprodukt dieser Domänen beschreibt aktuell zulässige Variablenbelegungen.

Definition 3.3.10 (induzierte Variablenbelegung) Ein ACS Constraint $c = ((v_1, ..., v_n), T, acs, z, E, \overline{h})$ *induziert* eine Variablenbelegung $(d_1, ..., d_n)$ von $(v_1, ..., v_n)$, wenn für alle Anweisungen v_i`.setDomain`$(\overline{p}_i)$ in T gilt: $d_i \in \mathsf{dom}(\overline{p}_i) \cap \mathsf{dom}(v_i)$. c induziert eine Menge von Variablenbelegungen B, wenn es jedes $\overline{d} \in B$ induziert. Man schreibt dann $c \models (d_1, ..., d_n)$ bzw. $c \models B$.

Bemerkung 3.3.6
Um die deklarative Semantik der Constraints umzusetzen, sind die Propagationsmethoden so implementiert, daß gilt $c \models \mathsf{sem}(c)$

Die Ausführungsreihenfolge der Propagationsmethoden bzw. die Reihenfolge der Programmanweisungen zur Domäneneinschränkung in den Propagationsmethoden spielt bei der Konstruktion der induzierten Variablenbelegung keine Rolle, weil in Def. 3.3.10 *alle* solchen Anweisungen erfüllt sein müssen. Dieser Fixpunkt der Domäneneinschränkungen wird in der Praxis durch mehrmalige Ausführung der Propagationsmethoden innerhalb der Constraintpropagation erreicht. Weil die Anweisungen zur Domänenreduktion eines Constraints selbst unveränderlich sind, ist die Menge der induzierten Variablenbelegungen eines Constraints daher ausschließlich abhängig von den Domänen der Variablen vor der Ausführung der Propagationsmethoden. Daher definiert jedes ACS Constraint eine Abbildung in den aktuellen Domänen der Variablen.

Definition 3.3.11 (induzierte Propagationsfunktion) Ein ACS Constraint $c = ((v_1, ..., v_n), T, acs, z, E, \overline{h})$ *induziert* eine Propagationsfunktion $\mathsf{prop} : \mathcal{P}ow(D_1 \times ... \times D_n) \to \mathcal{P}ow(D_1 \times ... \times D_n)$, wenn $\forall D \subseteq D_1 \times ... \times D_n$: $\mathsf{prop}(D) = D' \Rightarrow c \models D'$ gilt.

Bemerkung 3.3.7
Die Möglichkeit Constraints zurückzunehmen und damit die Variablendomänen zu erweitern, wird in Propagationsfunktionen nicht betrachtet. Der Begriff der induzierten Propagationsfunktion bezieht sich also nicht auf die tatsächlich stattfindende Veränderung der Variablendomänen, sondern lediglich auf die Definitionen von Variablendomänen in Propagationsfunktionen.

Lemma 3.3.1 Jede constraintfreie ACS Propagationsmethode m induziert eine eindeutige Propagationsfunktion prop_m.

Beweis:
Sei m eine beliebige ACS Propagationsmethode mit $\mathsf{vars}(m) = \{v_1, ..., v_n\}$. Laut Def. 3.3.6 und wegen der Constraintfreiheit kann m ausschließlich durch Definition von Variablendomänen (Def. 3.3.5.5) Einfluß auf $(\mathsf{dom}(v_1), ..., \mathsf{dom}(v_n))$ nehmen. Zu zeigen ist also, daß eine Sequenz von Anweisungen der Form `Entail=`v_i`.setDomain(`$\overline{p}$`)` eine eindeutige Propagationsfunktion induziert.

Für eine Anweisung gilt:

$$
\begin{gathered}
\texttt{Entail} = v_i\texttt{.setDomain(}\overline{p}\texttt{)} \\
\overset{Def.3.3.6.6}{\Rightarrow} \\
v_i = (\mathsf{dom}^{-1}(\mathsf{dom}(v_i) \cap \mathsf{dom}(\overline{p})), C) \wedge c = (\overline{v}, T, z, E, \overline{h}) \\
\overset{Lem3.2.1}{\Rightarrow} \\
v_i = (\mathsf{dom}^{-1}(\mathsf{dom}(v_i) \cap \mathsf{dom}(\overline{p})), C) \\
\wedge \mathsf{sem}(c) \subseteq \mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_i) \times \ldots \times \mathsf{dom}(v_n) \\
\Rightarrow \\
\mathsf{sem}(c) \subseteq \mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_i) \cap \mathsf{dom}(\overline{p}) \times \ldots \times \mathsf{dom}(v_n) \\
\Rightarrow \\
f_{v_i} : \mathcal{P}ow(\mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_n)) \rightarrow \mathcal{P}ow(\mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_n)) \\
\text{ist eindeutig definiert durch} \\
f_{v_i}(\mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_i) \times \ldots \times \mathsf{dom}(v_n)) \\
= \mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_i) \cap \mathsf{dom}(\overline{p}) \times \ldots \times \mathsf{dom}(v_n)
\end{gathered}
$$

Die Propagationsfunktion
$\mathsf{prop}_m : \mathcal{P}ow(\mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_n)) \rightarrow \mathcal{P}ow(\mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_n))$
für alle Domänendefinitionen in m ist dann eindeutig definiert durch
$\mathsf{prop}_m = \bigcap_{v \in \overline{v}} f_v$.

□

Damit ist gewährleistet, daß man ACS Propagationsmethoden zur Implementierung der Propagatoren in ACS Constraints (Def. 3.2.5) verwenden kann. Nun kann man zeigen, daß die Domäneneinschränkung constraintfreier Propagationsmethoden nicht zu viele Werte aus den Variablendomänen entfernen. Diesen Zusammenhang der deklarativen Semantik eines Constraints und seiner Implementierung als ACS Constraint habe ich in Lemma 3.2.1 bereits beschrieben. Die Idee ist, daß die ACS Constraints deklarativ genau so gedeutet werden, wie ihre Propagationsmethoden implementiert sind.

Lemma 3.3.2 Jede constraintfreie ACS Propagationsmethode m induziert eine eindeutige Domänenreduktionsfunktion prop_m.

Beweis:
Wegen Lemma 3.3.1 ist prop_m aus dem Beweis des Lemmas eine eindeutig bestimmte Propagationsfunktion (zu 1.)). Zu zeigen bleibt 2.), daß keine Lösungen verloren gehen:

Sei $c = ((v_1, \ldots, v_n), \{(m, w), \ldots\}, acs, z, E, \overline{h})$ beliebig, wegen Lemma 3.2.1 ist dann $\mathsf{sem}(c) \subseteq \mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_n)$ definiert.

1.) Es gilt
$\mathsf{sem}(c) \cap \mathsf{prop}_m(\mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_n)) \subseteq \mathsf{sem}(c) \cap \mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_n)$,
wegen Lemma 3.3.1 und Def. 2.2.1.

2.) Zu zeigen ist :
$\mathsf{sem}(c) \cap \mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n) \subseteq \mathsf{sem}(c) \cap \mathsf{prop}_m(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n))$

$$\text{Sei } (d_1, ..., d_n) \in sem(c) \cap \mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n) \text{ beliebig}$$
$$\Rightarrow$$
$$(d_1, ..., d_n) \in sem(c)$$
$$\overset{Lem.3.2.1,3.3.1}{\Rightarrow}$$
$$(d_1, ..., d_n) \in sem(c) \wedge (d_1, ..., d_n) \in \mathsf{prop}_m(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n))$$
$$\Rightarrow$$
$$(d_1, ..., d_n) \in sem(c) \cap \mathsf{prop}_m(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n))$$

Wegen 1.) und 2.) gilt
$sem(c) \cap \mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n) = sem(c) \cap \mathsf{prop}_m(\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n))$.
Somit ist prop_m eine Domänenreduktionsfunktion nach Definition 2.4.3.

□

Bemerkung 3.3.8
Die Semantik eines ACS Constraints wird also nicht durch seine deklarative Beschreibung oder durch Aufzählung gültiger Tupel definiert, sondern aus den Propagationsmethoden abgeleitet. In der Praxis ist es dann die Aufgabe der Programmiererin die Propagationsmethoden so zu implementieren, daß die gewünschte deklarative Semantik umgesetzt wird. So wird beim ACS, wie in den meisten constraintverarbeitenden Systemen, aus Effizienzgründen eine operationale Semantik eingesetzt, die von der deklarativen Semantik stark abweicht.

Um auch nicht-constraintfreie ACS Constraints theoretisch behandeln zu können, erweitere ich den bekannten Begriff der Domänenreduktionsfunktion [Apt98], so daß auch verschachtelte Constraints abgesetzt werden können (Def. 3.3.5.2). Dieser Schritt ist eine einfache Erweiterung der Funktion eines Constraints c um die Einschränkungen, die in Constraints aus der Menge $\mathsf{builtin}(c)$ folgen.

Definition 3.3.12 (DRF-Menge) Sei m eine ACS Propagationsmethode mit der endlichen Menge von Hilfsconstraints $\mathsf{builtin}(m)$ gegeben, dann ist die *DRF-Menge* von m definiert durch:
$\mathsf{prop}_m^+ := \begin{cases} \{\mathsf{prop}_m\}, \text{ falls } \mathsf{builtin}(m) = \emptyset \\ \{\mathsf{prop}_m\} \cup \bigcup_{c' \in \mathsf{builtin}(m)} \mathsf{prop}_{c'}^+, \text{ sonst.} \end{cases}$

Bemerkung 3.3.9
Damit ist auch zu jedem Constraint eine eindeutige DRF-Menge als Menge einfacher Domänenreduktionsfunktionen bestimmt.

Werden in einer Anwendung Propagationsmethoden verwendet, die durch unendliche Ketten abgesetzter Constraints implementiert sind, ist die Propagationsmethode nicht wohldefiniert. Daher wird in Definition 3.3.12 vorausgesetzt, daß die Akkumuluation der Hilfsconstraints endlich ist. Diese Wohldefiniertheit von Propagationsmethoden erweitere ich nun als weitere Eigenschaft von Constraintprogrammen.

Definition 3.3.13 (beschränkt) Eine ACS Propagationsmethode m heißt *beschränkt*, wenn $\mathsf{builtin}(m)$ endlich ist. Ein Constraint heißt beschränkt, wenn all seine Propagationsmethoden beschränkt sind. Ein Constraintprogramm P heißt *beschränkt*, wenn alle $c \in \mathsf{constraints}(P)$ beschränkt sind.

Fallstudie, Teil 7
Das Constraint `Disjoint` *aus Teil 6 der Fallstudie zu* $\mathcal{M}$*Plan ist beschränkt, weil die einzigen Hilfsconstraints die es verwendet, genau zwei* `Vor`*-Constraints sind. Die DRF-Menge von* `Disjoint` *berechnet sich durch die Domänenreduktion der Hilfsconstraints und entspricht somit dem punktweisen Durchschnitt der Propagationsfunktion "vor" aus Teil 4 unserer Fallstudie. Dies ist der Fall, weil das ACS Constraint* `Vor` *laut Teil 6 der Fallstudie genau die Propagationsfunktion "vor" induziert.*

3.4 Constraintpropagation

Zusammenfassung. In diesem Abschnitt wird die Semantik eines additiven Constraintprogramms als Fixpunkt der chaotischen Iteration der durch die abgesetzten Propagationsmethoden induzierten Domänenreduktionsfunktionen definiert. Es wird gezeigt, daß das ACS Ausführungsmodell diese Semantik operational berechnet. Dazu wird der Begriff des Zustands für ACS Constraintpropagation definiert und die Eigenschaften eines Ergebnis-Zustands angegeben. Außerdem wird beschrieben, unter welchen Voraussetzungen die deklarative Semantik bzw. ein Ergebnis der Programmabarbeitung existiert.

Durch das Absetzen von Constraints (Def.3.3.1.4) werden beim ACS Propagationsmethoden in einem Constraintlöser gepuffert. Den erwünschten Effekt des Absetzens, nämlich das Erkennen von Inkonsistenzen bzw. die Reduktion der Variablendomänen, erzielt der Solver erst nach ein- oder mehrmaliger Ausführung der Propagationsmethoden des Constraints. Durch die Anweisungen in den Propagationsmethoden wird Expertenwissen über die Constraintdomäne, das in den Methoden implementiert ist, auf konkrete aktuelle Variablendomänen angewandt. Daraus kann eine Einschränkung des Suchraums entstehen, aus der sich wiederum andere Einschränkungen mit anderen Constraints ableiten lassen. Der gewünschte Effekt einer Anweisung zum Absetzen eines Constraints in einem Constraintprogramm ist diese Einschränkung der Variablendomänen im gesamten induzierten Constraintnetzwerk des Programms. Außerdem können durch die Propagation von Wissen frühzeitig Inkonsistenzen erkannt und dann auch verarbeitet werden.

3.4.1 Chaotische Iteration in ACS

Die Ergebnisse aus [Apt98], die ich in Abschnitt 2.4 zusammengefaßt und für Propagationsfunktionen konkretisiert habe, werden nun in Zusammenhang mit ACS Constraints gesetzt. Durch die Möglichkeit, innerhalb von Propagationsmethoden andere Constraints absetzten zu können, ergibt sich für die dadurch induzierten Propagationsfunktionen ein Seiteneffekt. Es werden nämlich neben der Domänenreduktion durch die Funktion weitere Constraints abgesetzt, d.h.

weitere Propagationsfunktionen in den Iterationsprozeß eingefügt. Durch die Definition von DRF-Mengen (Def. 3.3.12) werden die Funktionen, die als Seiteneffekt aufgerufen werden können, mit betrachtet. Nicht-constraintfreie Propagationsmethoden induzieren anstatt einer Domänenreduktionsfunktion eine Menge solcher Funktionen (Bem. 3.3.9). Durch die Voraussetzung, daß die Propagationsmethoden wohldefiniert sind, daß sie also nur endlich viele (und wohldefinierte) Hilfsconstraints verwenden, ergibt sich, daß man auch aus solchen verschachtelten Constraintaufrufen eine eindeutig bestimmte Menge von Propagations- bzw Domänenreduktionsfunktionen ableiten kann.

Lemma 3.4.1 Jedes beschränkte ACS Constraint induziert eine eindeutige DRF-Menge.

Beweis:
Folgt wegen Lemma 3.3.2 und der Monotonie der Mengenvereinigung direkt aus Def. 3.3.12.

□

Durch Mengenvereinigung kann man die induzierten Domänenreduktionsfunktionen eines Programms ableiten.

Lemma 3.4.2 Jedes konsistente und beschränkte Constraintprogramm induziert eine endliche Menge von Domänenreduktionsfunktionen.

Beweis:
Sei ein beliebiges beschränktes Constraintprogramm P gegeben, dann ist $\{c_1, ..., c_n\} := \mathsf{constraints}(P)$ laut Def. 3.3.2 eindeutig bestimmt und laut Bem. 3.3.1 endlich. Aus Lemma 3.4.1 folgt, daß jedes $c_i \in \mathsf{constraints}(P)$ eine DRF-Menge D_i induziert. Weil P laut Voraussetzung beschränkt ist, folgt aus Def. 3.3.13, daß für $1 \leq i \leq n$ gilt, daß D_i endlich ist. Dadurch ist die Menge $\bigcup_{1 \leq i \leq n}(\bigcup D_i)$, die von P induzierte endliche Menge von Domänenreduktionsfunktionen, eindeutig bestimmt.

□

Bemerkung 3.4.1
Damit ist gezeigt, daß der Prozeß der Chaotischen Iteration nicht durch die Möglichkeit in den Domänenreduktionsfunktionen weitere Constraints abzusetzen, gestört wird. Dabei muß lediglich sichergestellt sein, daß dieses Absetzen keine rekursiven Constraintaufrufe beinhaltet. Diese Eigenschaft ist gesichert, wenn die DRF-Menge jeder verwendeten Propagationsmethode endlich ist.

Aus jedem beliebigen Constraintprogramm läßt sich also eine endliche Menge von Domänenreduktionsfunktionen ableiten. Die Funktionen bilden dabei jeweils Mengen von Belegungen der Variablen aufeinander ab, die in dem induzierenden Constraint verwendet werden. All diese Variablen können in jedem Constraintprogramm P in einem Constraintschema zusammengefaßt und geordnet werden. Man kann z.B. alle Elemente der Menge $\mathsf{vars}(P)$ in einem Vektor zusammenfassen und erhält ein geeignetes Constraintschema. Die induzierten Domänenreduktionsfunktionen können dann auf dieses Schema erweitert werden, indem sie an allen undefinierten Stellen (also den Domänen von Variablen, die nur in anderen Constraints vorkommen) als identische Abbildung definiert werden. Damit ist eine gemeinsame Domäne für alle Domänenreduktionsfunktionen eines Programms definiert. Diese Domäne ist durch eine Menge von Variablenbelegungen gegeben, wie sie in Def. 2.2.1 und Def. 2.4.3 angegeben wird. Wegen der Lemmata 2.4.2 und 2.4.1 ist damit die chaotische Iteration der induzierten Domänenreduktionsfunktionen eines Constraintprogramms definiert. Wie es in Bemerkung 2.4.2 bereits beschrieben wurde, sind alle Voraussetzungen für den Domänenreduktionssatz (Satz 1) damit erfüllt. Das dadurch eindeutig bestimmte Ergebnis der (unendlichen) Iteration betrachte ich als intendierte denotationale Semantik [Mos90] eines Constraintprogramms.

Definition 3.4.1 (Denotationale Semantik) Gegeben sei ein beschränktes und konsistentes Constraintprogramm P, das die DRF-Menge F induziert und das Constraintschema $\langle v_1, ..., v_n \rangle$ generiert. Dann ist die *denotationale Semantik* von P der Grenzwert der chaotischen Iteration von F^+ des Wertes $\mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n)$, man schreibt: $\mathsf{sem}(P) = \widehat{F}$.

3.4.2 Ausführung der Constraintlöser

Bei der Ausführung von Constraintprogrammen verwaltet jeder definierte Constraintlöser seinen eigenen Aufgabenpuffer. Dort werden Referenzen auf Propagationsmethoden gehalten, die später ausgeführt werden. In meiner ACS-Implementierung J.CP wird für jeden Constraintlöser ein nebenläufiger *thread* gestartet, in dem (endlos) anstehende Arbeitsaufgaben erfüllt werden. In diesem *thread* wird dazu der Code aus Abb. 3.3 ausgeführt.

```
1  while true do
2      if z = inconsistent                     // Inkonsistenz ist aufgetreten
3        then execute(ρ);                      //Inkonsistenz beseitigen
4             z = zuv(A)                       // Zuverlässigkeit neu berechnen
5        else if (A = ∅)                       // Warten auf neue Aufgaben
6             then  warte bis A ≠ ∅;
7             else                             // A = ⟨t1, t2, ..., tn⟩
8                A := ⟨t2, ..., tn⟩;
9                execute(t1)                   // Aufgabe erledigen
10               inc(z);                       // Zuverlässigkeit erhöhen
11           fi
12     fi
13 end
```

Abbildung 3.3: Die endlose Abarbeitung von Aufgaben im Asynchronen Constraintlöser $acs = (z, A, \rho)$. Zur Ausführung der Aufgaben (Zeile 9) wird entweder eine Propagationsmethode (Def. 3.3.6) oder der Algorithmus zur Constraintrücknahme (Abschnitt 3.5) ausgeführt, wie es in Abb. 3.4 dargestellt wird. Die Ausführung der Reparaturoperation (Zeile 3) kann z.B. in Algorithmen zur Constraintrücknahme umgesetzt werden.

Wenn eine Aufgabe aus dem Puffer eines Constraintlösers zur Bearbeitung entnommen wird, so wird die jeweilige Aufgabe direkt ausgeführt. Bei Propagatoren aus Constraints, aus denen vorher noch keine Propagationsmethoden ausgeführt wurden, wird vor der Ausführung eine Momentaufnahme der aktuellen Domänen der Variablen des Constraint als Domänenurbild abgespeichert. Diese dienen bei der Rücknahme von Constraints zur Konstruktion von erweiterten Variablendomänen.

```
funct execute(task)
  if task = post(c,p)
     mit c = ((v_1, ..., v_m), {(f_1, p_1), ..., (f_n, p_n)}, acs, active, E, h̄)
    then if h̄ == null              // die erste Ausführung des constraints
           then h̄ := (d_1, ..., d_n) mit v_i = (d_i, C)
         fi
         run(f_i) mit p_i = p        //ein aktives Constraint ausführen
     else if task = retract(c)       //ein Constraint zurücknehmen
           then remove(c)
         fi
  fi
end
```

Abbildung 3.4: Ausführung von Aufgaben durch den Constraintlöser. Die Ausführung von $run(f_i)$ ist in Def. 3.3.6 beschrieben. Die Ausführung von $remove(c)$ wird in Kapitel 3.5 und insbesondere in Def. 3.5.4 beschrieben werden.

Da die Ausführung des Constraintlösers nebenläufig zur Ausführung des Constraintprogramms, in dem Constraints abgesetzt werden, stattfindet, kann man nur schwer einen globalen Zustandsbegriff finden, der beide Komponenten beschreibt. Da solche globalen Zustände beim ACS aber nicht gebraucht werden bzw. verhindert werden sollen, um Verteilung zu unterstützen, definiere ich einen Zustandsbegriff, der sich ausschließlich auf die Abarbeitung der Aufgaben in den Constraintlösern bezieht. Um die Behandlung der abgesetzten Constraints in (imperativen) Constraintprogrammen zu berücksichtigen, definiere ich den Zustand des Programms durch die für die Propagation relevanten Programmvariablen. Dieser Zustand kann als Teil des Programmzustands verstanden werden, den das jeweilige imperative Programm in dem Moment hat. Es handelt sich also um eine Projektion der für das ACS Ausführungsmodell relevanten Teile aus dem allgemeinen Zustand für imperative Programme (z.B. [Hoa69]). Im Bezug auf die Ausführung von Propagationsmethoden ist für den Stand der Berechnung eines Constraintprogramms relevant, welches die aktuellen Variablendomänen sind und welche Aufgaben, die diese Domänen verändern könnten, noch anstehen.

Definition 3.4.2 (Zustand) Der *Zustand* eines Constraintprogramms P mit dem generierten Constraintschema $\langle v_1, ..., v_n \rangle$ und $\mathsf{solvers}(P) = \{(z_1, A_1, \rho_1), ..., (z_m, A_m, \rho_m)\}$ ist definiert durch $(\langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n) \rangle, \{A_1, ..., A_m\})$. Die Menge aller a posteriori feststellbaren Zustände von P heißt $\mathsf{states}(P)$.

Da Constraintprogramme von sich aus nicht terminieren (vgl. Abb. 3.3), kann man aus ihnen auch nicht allgemein einen Endzustand bzw. ein Ergebnis ableiten. Dies wurde bereits am Ende von Abschnitt 2.4 in Zusammenhang mit kontinuierlichen Constraintproblemen diskutiert. Durch die Tatsache, daß potentiell immer wieder neue Constraints in laufenden Solvern abgesetzt und damit in das bestehende Constraintnetz eingefügt werden können, kann man das Ergebnis der Berechnungen des Constraintlösers nur zu bestimmten Zeitpunkten betrachten. Geht man davon aus, daß ein geschlossenes CSP eingegeben wird und gelöst werden soll, so kann man das Ergebnis bzw. die Lösung des CSP frühestens dann in den Domänen der Variablen ablesen, wenn die Propagation aller Constraints (insbesondere auch von Instantiierungen der Variablen) abgeschlossen ist. Dieser Zustand ist daran zu erkennen, daß keine weiteren Aufgaben in den Puffern der definierten Solver anstehen. Da auch nur Implementierungen solcher abgeschlossenen CSPs eine denotationale Semantik haben (zur chaotischen Iteration werden endliche Constraintmengen vorausgesetzt), betrachte ich die Variablendomänen in dem Zustand, in dem keine Aufgaben mehr anstehen, als Ergebnis eines Constraintprogramms in ACS. Die Bearbeitung kontinuierlicher Constraintprobleme ist mit ACS zwar auch möglich, wie sich z.B. bei der Implementierung von $\mathcal{M}\mathsf{Plan}$ gezeigt hat, hierzu müßte aber eine andere (temporale) Semantik definiert werden.

Definition 3.4.3 (Ergebnis) Das *Ergebnis* eines konsistenten Constraintprogramms P (geschrieben $\mathsf{res}(P)$) sind die Variablendomänen im Zustand von P wenn nach der Ausführung von P für alle asynchronen Constraintlöser $acs = (z, A, \rho) \in \mathsf{solvers}(P)$ gilt $z = \top$, bzw. $A = \langle\rangle$.

3.4.3 Propagation und Terminierung

Wenn eine Variablendomäne in ACS eingeschränkt wird, werden nach Def. 3.3.6 bzw. Abb. 3.2 alle Constraints, die aus dieser Einschränkung weitere Einschränkungen anderer Variablen inferieren könnten, erneut abgesetzt. Es kann also bei jedem Constraint hyper-arc-Konsistenz (Def. 2.2.2) erzwungen werden. So viel Propagation wird in den meisten Constraintlösern (z.B. SICStus [COC97] oder ILOG [ILOG]) aus Effizienzgründen aber normalerweise nicht durchgeführt. Meist wird nur arc-Konsistenz durch *look-ahead* Propagation berechnet. Beim ACS kann auch ohne diese Einschränkung effizient propagiert werden, weil durch die Pufferung der Arbeitsaufgaben solche zeitaufwendigen oder wenig erfolgversprechenden Berechnungen zurückgestellt werden können. Die Herstellung der Konsistenz von n-ären Constraints, also hyper-arc-Konsistenz, kann z.B. immer nach der Propagation der binären Constraints ausgeführt werden, so daß man bereits während der laufenden Propagation ein gewisses Zuverlässigkeits-level erreicht hat. Wenn diese Zuverlässigkeit ausreicht, wenn also die Konsistenz binärer Constraints aus Zeitgründen ausreichen muß, kann man die abgeleiteten Variablendomänen dann bereits lesen und im Constraintprogramm weiter verwenden. Die Konsistenz der n-ären Constraints wird in diesem Fall erst dann hergestellt, wenn keine Aufgaben binärer Constraints in den Puffern der Constraintlöser anstehen. Beim ACS kann jeder Propagationsmethode eine Wichtigkeit zugeordnet werden (Def. 3.2.1), so daß bei der Constraintverarbeitung auch aufwendige Inferenzschritte bei Bedarf durchgeführt werden können. Sollte die Domäne einer Variablen nur noch einen Wert enthalten, so sollte dies bei der Propagation besonders berücksichtigt werden (Determination [FA97]). So wird in meiner ACS-Implementierung $\mathsf{J.CP}$ ein n-äres Constraint, bei dem eine Variable bereits instantiiert wurde, als $(n-1)$-äres Constraint im Puffer eines Constraintlösers eingeordnet und dadurch früher erneut ausgeführt.

Im Normalfall wird ein CSP immer solange bearbeitet, bis man eine Lösung gefunden hat, und keine Propagationsaufgaben mehr anstehen. Weil in den Propagationsmethoden ja außer den Domäneneinschränkungen auch Konsistenztests durchgeführt werden, kann man den Zustand der Solver nur dann als Ergebnis betrachten, wenn keine Aufgaben mehr anstehen (vgl. Def. 3.4.3). Zu zeigen bleibt, daß man beim ACS einen solchen Zustand auch erreicht. Da die Propagationsmethoden beliebige imperative Programme sein können, muß dazu vorausgesetzt werden, daß der Programmierer hier nur terminierende Algorithmen implementiert hat. Außerdem dürfen keine endlosen Ketten von Hilfsconstraints, wie sie z.B. durch rekursives Erzeugen und Absetzen von Constraints entstehen können, implementiert wurden.

Lemma 3.4.3 Jedes additive und konsistente Constraintprogramm P hat ein Ergebnis, wenn für alle $(\mathsf{prop}, w) \in \mathsf{propagators}(P)$ gilt:

1. prop und alle $f \in \mathsf{builtin}(\mathsf{prop})$ terminieren
2. $\mathsf{builtin}(\mathsf{prop})$ ist endlich.

Beweis:
Wegen Voraussetzung 1. ist die Ausführung (Def. 3.3.6) aller $f \in M = \mathsf{propagators}(P) \cup \bigcup_{1 \leq i \leq n} \mathsf{builtin}(f_i)$ endlich.

Ich zeige, daß bei der Ausführung von P gemäß Def. 3.3.4.4 und Def. 3.3.6.2,.5 jede Propagationsmethode $f \in M$ nur endlich oft in den Aufgabenpuffer A eines $acs \in \mathsf{solvers}(P)$ eingefügt wird.

1. Initiales Einfügen gemäß Def. 3.3.4.4:
   $\forall f \in \mathsf{propagators}(P) : \exists!(z, A, \rho) \in \mathsf{solvers}(P) : f \in \varepsilon(A)$
   Jedes $f \in \mathsf{propagators}(P)$ wird genau einmal eingefügt.
2. Einfügen von Hilfsconstraints gemäß Def.3.3.6.2:
   $\forall f \in \mathsf{propagators}(P) : \forall f' \in \mathsf{builtin}(f) : \exists!(z, A, \rho) \in \mathsf{solvers}(P) : f' \in \varepsilon(A)$
   Jedes $f' \in \mathsf{builtin}(f)$ für jedes $f \in \mathsf{propagators}(P)$ wird gemäß Bemerkung 3.3.4 nur genau einmal eingefügt.
   Es gibt nur endlich viele f' wegen Voraussetzung 2..
3. Einfügen bei der Propagation gemäß Def.3.3.6.5:
   Propagationsmethoden C_{P_i} werden laut Def. 3.3.6.5 nur eingefügt, wenn sich die Domäne $\mathsf{dom}(p_1, ..., p_i, ..., p_n)$ einer Variablen $v = (\overline{p}, (C)_{P \in \{P_1, ..., P_n\}})$ bzgl. $\subset$ verkleinert. Jede Menge $C_{P_i} \subseteq \mathsf{constraints}(P)$ ist endlich, weil $\mathsf{constraints}(P)$ laut Bem. 3.3.1 endlich ist. Angenommen, eine Propagationsmethode f würde unendlich oft eingefügt werden, dann könnte dies zwei Gründe haben:
   (a) $\exists f' \in M : f' \in \mathsf{builtin}(f) \wedge f \in \mathsf{builtin}(f')$
   Dieser Fall ist ausgeschlossen, weil es laut Def. 3.3.6.1 keine Möglichkeit gibt, eine Referenz auf ein bekanntes Constraint an ein neues Constraint weiterzugeben (vgl. Bem. 3.3.4).
   (b) $\exists v = ((p_1, ..., p_i, ..., p_n), \{C_{P_1}, ..., \{f, ...\}_{P_i}, ..., C_{P_n}\}) \in \mathsf{vars}(P) : (M \models (p_1, ..., p_i', ..., p_n) \wedge \mathsf{dom}((p_1, ..., p_i', ..., p_n)) \subset \mathsf{dom}((p_1, ..., p_i, ..., p_n)))$

Da aber M endlich ist und alle $f \in M$ fixe Definitionen von Variablendomänen (Def. 3.3.5.5) haben, kann es keine Variable geben, deren Domäne aufgrund von M unendlich oft eingeschränkt werden kann.

Da jeder $acs \in \mathsf{solvers}(P)$ gemäß Abbildung 3.3 potentiell unendlich viele Propagationsmethoden ausführt und nur endlich viele Propagationsmethoden als Aufgaben gestellt werden, muß der Zustand erreicht werden, in dem gilt:
$\forall (z, A, \rho) \in \mathsf{solvers}(P) : A = \langle\rangle$ und somit laut Bem. 3.2.1 auch $\forall (z, A, \rho) \in \mathsf{solvers}(P) : z = \top$.
Damit kann das Ergebnis von P nach Def. 3.4.3 aus den Variablendomänen abgelesen werden.

□

Wenn ein Constraintprogramm ein Ergebnis hat, dann sind alle verwendeten Constraints so programmiert, daß sie terminieren. Daraus läßt sich ableiten, daß ein anderes Constraintprogramm, das die gleichen Constraints verwendet, auch terminiert. Bezüglich der Terminierung spielt also die Reihenfolge in der Constraints abgesetzt werden oder die initialen Variablendomänen oder andere Variationen des Constraintprogramms keine Rolle.

Lemma 3.4.4 Hat ein Constraintprogramm P ein Ergebnis, dann hat auch jedes Constraintprogramm P' ein Ergebnis, wenn $\mathsf{constraints}(P') \subseteq \mathsf{constraints}(P)$ gilt.

Beweis:
Analog zum Beweis von Lemma 3.4.3, denn es wird jeweils nur die Teilmenge von Propagationsmethoden in die jeweiligen asynchronen Constraintlöser eingefügt, die durch $\mathsf{constraints}(P')$ induziert werden. Die Ausführung dieser Constraints terminiert aber wegen $\mathsf{constraints}(P') \subseteq \mathsf{constraints}(P)$ und Lemma 3.4.1, falls die Ausführung der von $\mathsf{constraints}(P)$ induzierten Propagationsmethoden terminiert.

□

Die denotationale Semantik eines Constraintprogramms wird aus den Propagationsmethoden der verwendeten Constraints abgeleitet. Diese existiert also auch nur dann, wenn diese Methoden wohldefinierte Domänenreduktionsfunktionen induzieren, so daß die Existenz eines Ergebnisses als Voraussetzung für die Definiertheit der denotationalen Semantik angegeben werden kann.

Lemma 3.4.5 Wenn ein Constraintprogramm ein Ergebnis hat, dann existiert seine denotationale Semantik.

Beweis:
Es sei ein konsistentes Constraintprogramm P gegeben, das ein Ergebnis hat. Aus Bedingung 2 in Lemma 3.4.3 folgt dann, daß die Bedingungen in Def. 3.3.13 wegen Lemma 3.4.2 erfüllt sind und P somit beschränkt ist. Aus Lemma 3.4.2 folgt damit, daß P eine DRF-Menge induziert, die laut Lemma 3.4.3.2 endlich ist. Damit kann nach Def. 3.4.1 die deklarative Semantik von P durch Chaotische Iteration berechnet werden.

□

Fallstudie, Teil 8
Das Constraintprogramm $\mathcal{M}$Plan terminiert immer dann, wenn die Propagation aller abgesetzten Constraints abgeschlossen ist. Propagation wird immer dann ausgeführt, wenn sich eine Domäneneinschränkung einer Variablen ergibt. Auch wenn das Programm, also alle gestarteten Agenten, dann noch nicht terminiert haben, so repräsentieren die aktuellen Variablendomänen das Ergebnis des Systems. In diesem Fall ist durch die Übereinstimmung von operationaler und denotationaler Semantik gesichert, daß alle bereits festgelegten Termine überschneidungsfrei stattfinden, weil jeweils paarweise ein `Disjoint`*-Constraint abgesetzt wurde (Fallstudie Teil 6).*

3.4.4 Korrektheit und Vollständigkeit

Wenn ein Constraintprogramm ein Ergebnis liefert, dann sollte es der denotationalen Semantik des Programms entsprechen. Ich zeige also, daß das Absetzen von Constraints und die Propagation beim ACS genau die Variablendomänen berechnet, die auch der Fixpunkt der chaotischen Iteration der durch die Constraints induzierten Domänenreduktionsfunktionen ist. Falls ein Programm kein Ergebnis hat (weil es inkonsistent ist oder nicht terminiert), so ist auch die denotationale Semantik des Programms laut Lemma 3.4.5 nicht definiert, so daß keine Aussage über Korrektheit und Vollständigkeit getroffen werden kann.

Lemma 3.4.6 Die Propagation in ACS ist korrekt, d.h. für jedes additive Constraintprogramm P mit dem Ergebnis $\mathsf{res}(P) = \langle D_1, ..., D_n \rangle$ gilt: $\mathsf{sem}(P) \subseteq D_1 \times ... \times D_n$.

Beweis:
Es gilt wegen Def. 3.4.3, 3.4.2 und Def. 3.4.1:

$$\exists D'_1, ..., D'_n : \forall i \in 1..n : D_i \subseteq D'_i \wedge \mathsf{sem}(P) \subseteq D'_1 \times ... \times D'_n \tag{3.1}$$

Das Lemma wird indirekt bewiesen:
Annahme: $\neg(\mathsf{sem}(P) \subseteq D_1 \times ... \times D_n)$

$$\Rightarrow$$
$$\exists (d_1, ..., d_n) \in \mathsf{sem}(P) : (d_1, ..., d_n) \notin D_1 \times ... \times D_n$$
$$\Rightarrow$$
$$\exists (d_1, ..., d_i, ..., d_n) \in \mathsf{sem}(P) : \exists i \in 1..n : d_i \notin D_i$$
$$\overset{Formel (3.1)}{\Rightarrow}$$
$$\exists (d_1, ..., d_i, ..., d_n) \in \mathsf{sem}(P), d_i \notin D_i : d_i \in D'_i$$
$$\overset{Def. 3.2.6}{\Rightarrow}$$
$$\exists (d_1, ..., d_i, ..., d_n) \in \mathsf{sem}(P), d_i \notin D_i : \exists (f, w) \in \mathsf{propagators}(P) : d_i \in \mathsf{red}(f)$$

$$\Rightarrow$$

$$\exists(d_1,...,d_i,...,d_n) \in \mathsf{sem}(P), d_i \notin D_i, (f,w) \in \mathsf{propagators}(P), d_i \in \mathsf{red}(f) :$$
$$\exists D_i'' : f(D_1 \times ... \times D_i'' \times ... \times D_n) = D_1 \times ... \times D_i \times ... \times D_n \wedge d_i \in D_i'' \setminus D_i$$

$$\Rightarrow$$

$$\exists(d_1,...,d_i,...,d_n) \in \mathsf{sem}(P)(f,w) \in \mathsf{propagators}(P) :$$
$$f(D_1 \times ... \times D_i'' \times ... \times D_n) = D_1 \times ... \times D_i \times ... \times D_n \wedge D_i'' \supset D_i$$

$$\Rightarrow$$

$$\exists(d_1,...,d_i,...,d_n) \in \mathsf{sem}(P), (f,w) \in \mathsf{propagators}(P) :$$
$$f(D_1 \times ... \times D_i'' \times ... \times D_n) \subset D_1 \times ... \times D_i \times ... \times D_n$$

$$\overset{(d_1,...,d_i,...,d_n)\in\mathsf{sem}(P)}{\Rightarrow}$$

$$\exists(d_1,...,d_i,...,d_n) \in \mathsf{sem}(P), (f,w) \in \mathsf{propagators}(P) :$$
$$\mathsf{sem}(P) \cap f(D_1 \times ... \times D_i'' \times ... \times D_n) \subset \mathsf{sem}(P) \cap D_1 \times ... \times D_i'' \times ... \times D_n$$

$$\overset{Def.2.4.3}{\Rightarrow}$$

f ist keine Domänenreduktionsfunktion

$$\Rightarrow$$

Widerspruch zu Lemma 3.3.2

□

Lemma 3.4.7 Die Propagation in ACS ist vollständig, d.h. für jedes additive Constraintprogramm P mit dem Ergebnis $\mathsf{res}(P) = \langle D_1, ..., D_n\rangle$ gilt: $D_1 \times ... \times D_n \subseteq \mathsf{sem}(P)$.

Beweis:
Sei $F = \{f_1, ..., f_n\}$ die DRF-Menge, die P gemäß Lemma 3.4.2 induziert, so daß laut Definition 3.4.1

$$\mathsf{sem}(P) = \widehat{F} \tag{3.2}$$

gilt.

Sei $\mathsf{res}(P) = \langle D_1, ..., D_n\rangle$, dann gilt für jede Variablenbelegung:

$$(d_1,...,d_i,...,d_n) \in D_1 \times ... \times D_n$$

$$\overset{Def.3.4.3}{\Rightarrow}$$

$$\neg\exists \mathsf{prop} \in \mathsf{propagators}(P) : d_i \in \mathsf{red}(\mathsf{prop})$$

$$\overset{Lem.3.3.2}{\Rightarrow}$$

$$\neg\exists f \in F : d_i \in \mathsf{red}(f)$$

$$\overset{Satz1,Def.3.2.6}{\Rightarrow}$$

$$\neg\exists d_i \in D_i : (d_1,...,d_i,...,d_n) \notin \widehat{F}$$

$$\Rightarrow$$

$$\forall d_i \in D_i : (d_1,...,d_i,...,d_n) \in \widehat{F}$$

$$\overset{Formel(3.2)}{\Rightarrow}$$

$$\forall d_i \in D_i : (d_1, ..., d_i, ..., d_n) \in \mathsf{sem}(P)$$

$$\Rightarrow$$

$$D_1 \times ... \times D_n \subseteq \mathsf{sem}(P)$$

□

Weil das Ergebnis der chaotischen Iteration eindeutig bestimmt ist und genau dieses durch die Propagation in ACS erreicht wird, gilt das folgende Korollar.

Korollar 3.4.1 Zu jedem terminierendem Constraintprogramm P ist $\mathsf{res}(P)$ eindeutig bestimmt und entspricht seiner denotationalen Semantik.

3.5 Inverse Constraintpropagation

Zusammenfassung. Als Umkehroperation zum inkrementellen Absetzen von Constraints wird in diesem Abschnitt das inkrementelle Zurücknehmen von Constraints (*constraint retraction*) beschrieben. Wenn ein Constraint zurückgenommen wird, so sollen all seine Auswirkungen auf Variablen und andere Constraints rückgängig gemacht werden. Die beiden bekannten Ansätze für diese Aufgabe und mein Algorithmus werden beschrieben und verglichen. Im Gegensatz zu den bekannten Ansätzen verwendet *constraint retraction* in ACS keine globalen Datenstrukturen, so daß der Algorithmus auch für verteilte Anwendungen einsetzbar ist. Schließlich wird die Korrektheit des Algorithmus nachgewiesen und gezeigt, das er unter zulässigen Bedingungen terminiert.

Um NP-vollständige Probleme zu lösen, reichen die bekannten Algorithmen zur Herstellung von Konsistenz bzw. Constraintpropagation im Allgemeinen nicht aus [Wal75, MS98]. Daher muß man beim Lösen von CSPs oft nichtdeterministische Suchalgorithmen verwenden, um Lösungen zu finden. In klassischen CLP Systemen wie CHIP [Chip] oder SICStus [COC97] wird das Prologinhärente *backtracking* durch sogenannte *labelling*-Algorithmen gesteuert. Solche Algorithmen instantiieren einzelne Variablen, was durch die inkrementelle Verarbeitung von Constraints, also der Propagation der durch die Instantiierung eingeschränkten Variablendomäne, auf Korrektheit überprüft wird. Wird bei der Propagation eine Inkonsistenz entdeckt, so wird *backtracking* eingeleitet, was zu einer alternativen Instantiierung führt. Beim *backtracking* wird die Instantiierung der Variablen und alle damit verbundenen Auswirkungen im Constraintnetz durch das Wiederherstellen eines ehemaligen globalen Systemzustands rückgängig gemacht. In den meisten Constraintlösern wird zu diesem Zweck durch *trailing* oder *copying* Techniken (z.B. [Sch99]) die Geschichte der Variablendomänen im Zusammenhang mit den abgesetzten Constraints global abgespeichert.

Um die globale Verwaltung von Zuständen zu verhindern und damit auch in verteilten Systemen ehemalige Zustände herstellen zu können, wurden im Zusammenhang mit den Systemen Mozart [MOz, Sch00, Sch97] und Figaro

[Fig, CHN01] alternative Techniken entwickelt. Diese berechnen die gesuchten Zustände nicht durch rückwärtsgerichtete Navigation im Suchbaum, sondern durch Rekonstruktion des gesuchten Zustands von der Wurzel des Baumes aus. Die Idee, Fehlentscheidungen beim *labelling* durch Wiederherstellung ehemaliger Zustände rückgängig zu machen, wird aber auch in diesen Ansätzen verfolgt.

Alternativ zum Herstellen ehemaliger Zustände sind auch Suchalgorithmen denkbar, die eine Umkehroperation der Instantiierung verwenden, um Fehlentscheidungen rückgängig zu machen. Eine solche Umkehroperation ist Constraint *retraction* (z.B. [GCR99]). Das Zurücknehmen von Instantiierungen kann durch Constraint *retraction* im Unterschied zum *backtracking* unabhängig von der Reihenfolge in der sie abgesetzt wurden, erledigt werden. Weiterhin kann man mit Constraint *retraction* jedes beliebige Constraint, das abgesetzt wurde, zurücknehmen. Der Effekt der Operation ist, daß alle Auswirkungen, die das Constraint im aktuellen Constraintnetz hatte, entfernt werden. Es wird also ein Zustand berechnet, der entstanden wäre, wenn das zurückgenommene Constraint niemals abgesetzt worden wäre.

Ist in einem Constraintverarbeitenden System die Möglichkeit vorhanden, Constraints in der beschriebenen Weise zurückzunehmen, so ist die Verwaltung ehemaliger Zustände nicht mehr notwendig, so daß man hierzu auch keine globalen Datenstrukturen mehr benutzen muß. Fehlentscheidungen, die ja nur durch das Absetzen von Constraints im Solver gemacht werden können, werden durch Zurücknehmen des jeweiligen Constraints revidiert. Anschließend können dann alternative Constraints, wie z.B. Instantiierungen mit anderen Werten, innerhalb nicht-deterministischer Suchalgorithmen abgesetzt werden.

Aber nicht nur zur Implementierung von Suchalgorithmen kann Constraintrücknahem sehr gut eingesetzt werden. Bei dynamischen CSP [VS94, WF98, Bes91, DD98] können beliebige Constraints jederzeit inkrementell zurückgenommen werden. Insbesondere in interaktiven oder kontinuierlichen Anwendungen ist oft die Möglichkeit erforderlich, früher abgesetzte Constraints zu entfernen sehr wichtig, um nicht das gesamte Problem neu berechnen zu müssen. Insbesondere bei kontinuierlichen Anwendungen ist dies normalerweise nicht möglich. Aber auch bei relativ großen Problemen soll möglichst nicht immer das gesamte Problem neu berechnet werden müssen, um eine falsche bzw. veraltete Eingabe rückgängig zu machen. Im Terminplanungstool $\mathcal{M}$Plan wird zum Beispiel Constraint *retraction* immer dann beutzt, wenn ein Termin abgesagt wurde.

Fallstudie, Teil 9
Bereits bei der Erzeugung eines Terminvorschlags werden in $\mathcal{M}$Plan Constraints erzeugt und abgesetzt. Wenn dieser Termin nicht stattfindet, weil er z.B. von den anderen Teilnehmern abgesagt wurde sollten diese Constraints zurückgenommen werden um keine unnötigen Einschränkungen für andere Termine zu verursachen. Es ist bei diesem Problem nicht möglich, alle Berechnungen erneut auszuführen, weil es offen ist und keine zentrale Stelle existiert, an der alle gültigen Constraints bekannt sind. Selbst wenn alle Constraints durch Traversierung des Netzwerkes gesammelt werden könnten ist das Problem sicherlich zu groß, um es insgesamt erneut zu lösen. Außerdem brauchen Constraints, die ausschliesslich vergangenen Termine, oder solche, die nichts mit dem Abgesagten zu tun haben nicht betrachtet zu werden. Weil in keinem Constraintlöser die Möglichkeit zur Constraintrücknahme in verteilten Umgebungen existiert, konnte auch noch kein Werkzeug für dieses Problem hergestellt werden, in dem die

durch Absagen frei gewordenen Resourcen anderweitig genutzt werden können.

3.5.1 Inkrementelle Constraint *retraction*

Wenn ein Constraint zurückgenommen wird, soll im Constraintlöser ein Zustand erzeugt werden, der äquivalent zu dem Zustand ist, der der aktuelle Zustand wäre, wenn das Constraint nie abgesetzt worden wäre. Dies erfordert beim inkrementellen Constraintlösen einige Berechnungen, denn durch das Absetzen eines Constraints c wird das Constraintnetz folgendermaßen verändert:

1. Die Domänen der Variablen von c können eingeschränkt werden,
2. die Domänen anderer Variablen werden durch Propagation von c verändert,
3. bei der Ausführung später abgesetzter Constraints werden die so veränderten Domänen verwendet.

Um das Constraint zurückzunehmen, müssen diese Veränderungen rückgängig gemacht werden. Außerdem sollte ein zurückgenommenes Constraint aus dem System vollständig entfernt werden, so daß es keinen Speicher mehr belegt und nach der Rücknahme keine Auswirkungen mehr auf Variablen oder Constraints haben kann.

In nicht-inkrementellen Constraintlösern wie ILOG Solver [Ilog] oder JCL [JCL] ist das zurücknehmen von Constraints ohne aufwendige Reparaturoperationen möglich. Dort werden zu löschende Constraints lediglich aus dem CSP entfernt, das später gelöst wird. Es brauchen keinerlei Auswirkungen des Constraints rückgängig gemacht werden, weil es diese nicht gibt. Man muß in Anwendungen dieser Systeme für dynamische CSP nach jeder dynamischen Änderung das gesamte Problem neu lösen, bereits vorhandene Berechnungen können dabei nicht wieder benutzt werden. Solche Systeme haben (wegen ihrer nicht-inkrementalität) damit auch nicht die Vorteile, die ich in Abschnitt 3.1 beschrieben habe. Anwendungen, in denen CSPs interaktiv mit dem Benutzer aus der Propagation vorheriger Eingaben definiert werden, zum Beispiel erfordern neben dem inkrementellen Hinzufügen von Constraints auch das inkrementelle Entfernen fehlerhafter Eingaben.

Dennoch sollte die Möglichkeit, ein Constraint zurückzunehmen bevor es Auswirkungen auf das Constraintnetz hat, nicht ausgeschlossen werden. Dies ist beim ACS durch die Einordnung von *retraction*-Aufgaben an den Anfang der Puffer der Solver möglich, indem der *retraction*-Methode eine besonders hohe Wichtigkeit zugeordnet wird. Durch die asynchrone Ausführung kann ein Constraint beim ACS bereits zurückgenommen werden, bevor seine Propagation vollständig abgeschlossen ist bzw. überhaupt begonnen wurde. Dies kann in bestimmten Anwendungen, in denen wiederholt Constraints abgesetzt und zurückgenommen werden, zu erheblichen Laufzeitverbesserungen im Vergleich zu synchronen Constraintlösern führen, was in Abschnitt 7.1 genauer untersucht wird.

3.5.2 Bekannte Algorithmen und Ergebnisse

In der Literatur zum dynamischen Constraintlösen werden zwei inkrementelle Algorithmen zur Constraint *retraction* vorgestellt, die ich in diesem Abschnitt kurz beschreiben werde. Der *retraction*-Algorithmus zum ACS, der im nächsten Abschnitt vorgestellt wird, verwendet Konzepte dieser beiden Algorithmen um die Nachteile beider Techniken auszugleichen.

retraction in clp(FD,S)

In die frei erhältlichen CLP Sprache clp(FD) [CD96] wurde Constraint *retraction* implementiert [GCR99]. Die theoretischen Grundlagen und Ideen wurden zuvor in [CDR96] und zusammenfassend in [Geo99] dargestellt. In der aktuellen clp(FD)-Weiterentwicklung GNUProlog [Gnu] wurde aber die Möglichkeit zur Constraint *retraction* nicht weiter unterstützt, während in der "akademischeren" Weiterentwicklung clp(FD,S) [CLP] immer noch *retraction* zur Verfügung steht.

Diese Systeme sind CLP Systeme, die durch *trailing* Techniken ehemalige globale Zustände verwalten und systeminhärentes *backtracking* benutzen. Ziel der Integration von *retraction* ist in diesem Ansatz, einzelne Constraints aus dem Netz zu entfernen, ohne *alle* Aktionen, die nach dem Absetzen des Constraints ausgeführt wurden, nach der Rücknahme wiederholen zu müssen. Dies soll geschehen, ohne die Algorithmen zum Absetzten von Constraints und der Propagation zu verlangsamen. Es sollten also keine zusätzlichen Informationen abgespeichert oder zusätzlichen Berechnungen bei der Propagation durchgeführt werden. Die wesentliche Idee besteht darin, die Abhängigkeiten von Constraints, die in den Variablen abgespeichert sind (Def. 2.1.2 Punkt 2) zu nutzen, um die Constraints zu finden, die zur Berechnung der neuen Domänen notwendig sind. Um ein Constraint c zurückzunehmen, werden zunächst aus den Listen abhängiger Constraints in den Variablen durch Traversierung des Constraintnetzes alle Constraints gesammelt, die die Domäneneinschränkungen von c benutzt haben. Dann werden die Domänen aller Variablen dieser Constraints in den Zustand zurückversetzt, den sie vor dem Absetzen von c hatten. Jetzt werden alle gesammelten Constraints nochmals abgesetzt, um den neuen globalen Zustand zu berechnen. Der Algorithmus ist in Abb. 3.5 dargestellt.

$deactivate(c)$; // c als inaktive markieren
$restore(dom(X), c)$; // Domäne von X erweitern: Zustand von vor c
$S := constraints(X)$ // alle Constraint über X
foreach $c' = X'$ `in` $r' \in S$ **do**
  if $active(c')$ // falls c' nicht gelöscht wurde
    then $restore(dom(X'), c)$ // Domäne von X' erweitern
      $S := S \cup constraints(X')$ // alle Constraint über X'
  fi
end
$repost(S)$ // alle relevanten Constraints erneut absetzen

Abbildung 3.5: Der *retraction* Algorithmus aus clp(FD,S) [GCR99] für ein Constraint $c = X$ `in` r, das die aktuelle Domänen von X mit r schneidet. Solche Constraints sind die einzigen *builtin*-Constraints in clp(FD,S) [CD96].

Der Unterschied zum trivialen *retraction* Algorithmus in CLP, nämlich dem Rücksprung im Suchbaum vor das zurückzunehmende Constraint und der Wiederholung aller Aktionen nach dem Absetzen des Constraints, besteht also darin, daß nicht alle Variablendomänen erweitert und nur manche Constraints erneut abgesetzt werden. Dieser Gewinn muß allerdings durch die recht aufwendige Traversierung des Constraintnetzes zum Finden abhängiger Constraint bezahlt werden. In Laufzeituntersuchungen [GCR99] wurde aber trotzdem, vor allem in lichten Constraintnetzen, ein signifikanter Performanzgewinn gemessen.

Die Idee, die Abhängigkeiten der Eigenschaften der Variablendomänen von bestimmten Constraints, die in den Variablen ohnehin abgespeichert werden zu benutzen, um relevante Constraints zur Herstellung des gesuchten Zustands zu finden, habe ich bei ACS übernommen. Dieses Vorgehen dient in ACS zur Handhabung der Constraints, die nach dem zurückgenommenen Constraint abgesetzt wurden und eventuell fehlerhafte Domäneneinschränkungen zu weiteren Inferenzschritten verwenden (Punkt 3. auf Seite 65).

Adaptive Constraintverarbeitung mit CHR

In [Wol99, WGG00] wird ein Ausführungsmodell zur Verarbeitung von CHR Constraints beschrieben, das neben inkrementellem Hinzufügen von Constraints auch inkrementelles Zurücknehmen erlaubt. Im Unterschied zu [GCR99] wird aber schon während des Constraint-Absetzens und der Propagation Wissen gespeichert, das speziell zur Constraint *retraction* verwendet wird. Die erlaubten Constraints werden durch Constraint-Handling-Rules (CHR) [Frü98] inkrementell verarbeitet und in einem zentralen Constraintspeicher verwaltet.

Wenn ein neues Constraint abgesetzt wird, so erhält es einen Index und wird unter diesem Index im Constraintspeicher registriert. Bei der Propagation der Constraints wird dann jedes abgeleitete Constraint mit den Indizes der Constraints markiert, aus denen es gefolgert wurde. Diese Folgerungen (*entailment*) werden explizit als Constraints behandelt und auch indiziert und in den Constraintspeicher eingefügt. So werden Abhängigkeitsketten zwischen den abgesetzten Constraints und allen Folgerungen des Systems aufgebaut, die zur Constraint *retraction* verwendet werden. Soll eines der abgesetzten Constraints zurückgenommen werden, so werden alle Folgerungen aus diesem Constraint aus dem Constraintspeicher gelöscht. Es ergibt sich ein rekursiver Algorithmus, der einen Systemzustand herstellt, in dem in keinem (abgeleiteten) Constraint mehr der Index zurückgenommener Constraints referenziert wird. Die Variablendomänen werden in CHR nicht explizit in den Variablen abgespeichert, sondern durch Constraints beschrieben. Daher ist, im Gegensatz zum Algorithmus von Georget [GCR99], keine Neuerechnung der Domänen notwendig.

Die Idee, die Abhängigkeitsketten bei der Propagation abzuspeichern, habe ich in ACS auch verwendet. Der Unterschied zu [Wol99] ist aber, daß die Referenzen nicht rückwärtsgerichtet als Begründungen, sondern vorwärtsgerichtet als Konsequenzen, abgespeichert werden. Ein Constraint kann immer nur unter den Constraints Konsequenzen haben, die bereits bekannt waren, bevor es ausgeführt wurde. Dadurch können die Auswirkungen, die ein zurückzunehmendes Constraint auf Variablen anderer Constraints durch Propagation hat (Punkt 2 auf Seite 65), durch Betrachtung von Konsequenzen revidiert werden.

3.5.3 *Retraction* in ACS

Bei der Propagation von neu abgesetzten Constraints werden bereits bekannte Constraints nochmals abgesetzt, um das neue Wissen möglichst weit im Constraintnetzwerk zu verbreiten. Daraus enstehen Abhängigkeitsketten von Constraints, die gemäß Def. 3.3.6.6 bzw. Abb. 3.2 in den Constraints als direkte Konsequenzen gespeichert werden. Durch die Propagation über mehrere Variablen enstehen dabei auch indirekte Konsequenzen, die es bei der Betrachtung der Auswirkungen eines Constraints zu berücksichtigen gilt.

Definition 3.5.1 (Konsequenzen) Sei ein ACS Constraint $c = (\overline{v}, T, acs, active, E, \overline{h})$ gegeben, dann ist die Menge der *Konsequenzen* von c $\mathsf{cons}(c)$ definiert durch

- $E \subseteq \mathsf{cons}(c)$
- $\forall c' \in \mathsf{cons}(c).\mathsf{cons}(c') \subseteq \mathsf{cons}(c)$

Entsprechend werden die Konsequenzen einer Menge C von Constraints definiert durch $\mathsf{cons}(C) = \bigcup_{c \in C} \mathsf{cons}(c)$.

Bemerkung 3.5.1
Die Konsequenzen eines Constraints beinhalten diejenigen Constraints, die bei der Propagation einer Domäneneinschränkung aufgrund von c nochmals abgesetzt wurden (siehe Def. 3.3.5). Die Hilfsconstraints eines Constraints sind keine Konsequenzen, weil sie nicht durch Propagation erneut ausgeführt werden. Da bei der nochmaligen Ausführung dieser Constraints weitere Domäneneinschränkungen entstehen können, müssen die Konsequenzen rekursiv im Constraintnetz akkumuliert werden.

Lemma 3.5.1 Ein ACS Constraint hat nur endlich viele Konsequenzen.

Beweis:
Weil alle Konsequenzen in einer Menge akkumuliert werden und es laut Bem. 3.3.1 nur endlich viele Constraints gibt (die vor einem gegebenen Constraint abgesetzt wurden), ist folglich auch die Menge der Constraints, die Konsequenzen sind, endlich.

□

In Abb. 3.6 wird ein Programmfragment zur Akkumulation von Konsequenzen in einem Constraintnetz angegeben.

```
funct getAllConsequences(C)
  foreach c' = (v̄', T', acs', active, E', h̄') ∈ E \ C do
                                    //Für alle direkten Konsequenzen...
    C := C ∪ c'.getAllConsequences(C);
                                    //...Sammle alle indirekten Konsequenzen
  end
  return C ∪ E;
.
```

Abbildung 3.6: Rekursive Akkumulation aller gespeicherten Konsequenzen eines Constraints $c = (\overline{v}, T, acs, z, E, \overline{h})$.

Wie es in [GCR99] ausführlich untersucht wurde, hat ein Constraint c auch Auswirkungen auf Constraints, die nach ihm abgesetzt wurden. Das sind all die Constraints, die Variablendomänen , die direkt oder indirekt durch c eingeschränkt wurden, für weitere Inferenzschritte nutzen. Diese Constraints können aus den Einträgen abhängiger Constraints (Def.2.1.2 Punkt 2) in den Variablen abgeleitet werden. Da die abhängigen Constraints bei Domäneneinschränkungen immer hinten an die Listen angehängt werden (Def. 3.3.6.6), sind alle Constraints, die nach c in einer solchen Liste stehen gefährdet, durch Variableneinschränkungen von c falsche Folgerungen gemacht zu haben. In den Variablen dieser Constraints kann man mit dem gleichen Vorgehen Constraints finden, die indirekt die durch c eingeschränkten Variablendomänen zur Inferenz nutzen. Die Akkumulation all dieser Constraints nenne ich verwendende Constraints von c.

Definition 3.5.2 (verwendende Constraints) Sei ein Constraint $c = (\overline{v}, T, acs, active, E, \overline{h})$ gegeben, dann ist die Menge $\mathsf{verw}(c)$ der *verwendenden Constraints* von c gegeben durch

- $\forall v = (\overline{p}, \{C_{P_1}, ..., C_{P_n}\}) \in \varepsilon(\overline{v}) : \forall i \in 1..n : C_{P_i} = \langle ..., c, c_1, ..., c_m \rangle \Rightarrow c_1, ..., c_m \in \mathsf{verw}(c)$
- $c' \in \mathsf{verw}(c) \Rightarrow \mathsf{verw}(c') \subseteq \mathsf{verw}(c)$

Bemerkung 3.5.2
Die Menge der verwendenden Constraints eines Constraints c sind also all diejenigen Constraints, die nach c ausgeführt wurden und eine Domäneneigenschaft verwenden, die c verändert hat. Diese Constraints werden rekursiv akkumuliert. Für jedes Constraint ist aus der Perspektive der Variablen auch im asynchronen Fall immer entscheidbar, ob es vor oder nach einem anderen Constraint ausgeführt wurde, wenn der Zugriff (Def. 3.3.5.6) z.B. durch eine Schloßvariable synchronisiert wird.

Lemma 3.5.2 Jedes ACS Constraint hat endlich viele verwendende Constraints.

Beweis:
Analog zum Beweis von Lemma 3.5.1.

□

Die Akkumulation der verwendenden Constraints aus Def. 3.5.2 wird zur Veranschaulichung in Abb. 3.7 als Programmfragment angegeben.

Die Constraints, die wie es auf Seite 65 beschrieben wurde, aufgrund der Domäneneinschränkungen eines Constraints weitere Einschränkungen gemacht haben könnten, können jetzt in einer Menge zusammengefaßt werden. Diese Menge der betroffenen Constraints eines Constraints c enthält neben allen Konsequenzen und allen verwendenden Constraints von c auch die Konsequenzen der verwendenden Constraints und c selbst. In Abb. 3.8 wird an einem Constraintnetz exemplarisch dargestellt, wo die einzelnen Teilmengen der betroffenen Constraints zu finden sind.

Definition 3.5.3 (betroffene Constraints) Sei ein Constraint c gegeben, dann sind in der Menge $\mathsf{betr}(c) := \mathsf{cons}(c) \cup \mathsf{verw}(c) \cup \mathsf{cons}(\mathsf{verw}(c)) \cup \{c\}$ alle von c *betroffenen* Constraints enthalten.

Nun kann gezeigt werden, daß in der Menge der betroffenen Constraints eines abgesetzten Constraints c alle Constraints enthalten sind, die direkt oder indirekt Variableneinschränkungen aus c ableiten.

funct $getAllLaterPostings(C)$
  foreach $v = (\overline{p}, L) \in \varepsilon(\overline{v})$ **do**
    foreach $C_P = \langle c_1, ..., c_i, c, c_j, ..., c_n \rangle \in L$ **do** //jedes Merkmal
      $M := \{c_j, ..., c_n\};$ //sammle die späteren Constraints...
      foreach $actu = (\overline{v}', T', acs', active, E', \overline{h}') \in M \setminus C$ **do**
        $M := M \cup actu.getAllLaterPostings(C \cup M);$ //...rekursiv
      end
    end
  end
  return $M \cup C;$

Abbildung 3.7: Rekursive Akkumulation aller verwendenden Constraints eines Constraints $c = (\overline{v}, T, acs, z, E, \overline{h})$. Dabei gelten die Constraints im Parameter C als bereits untersucht und brauchen nicht weiter berücksichtigt zu werden.

Lemma 3.5.3 Seien die additiven Constraintprogramme P und P' mit dem gemeinsamen generierten Constraintschema $\langle v_1, ..., v_n \rangle$ gegeben, so daß $\mathsf{instr}(P) = \mathsf{instr}(P') \cup \{c.post(s)\}$ gilt. Sei weiter $\mathsf{res}(P) = \langle d_1, ..., d_n \rangle$ und $\mathsf{res}(P') = \langle d'_1, ..., d'_n \rangle$, dann gilt $\forall i \in 1..n : d_i \neq d'_i \Rightarrow \exists c' \in \mathsf{betr}(c) \cup \mathsf{builtin}(c) : v_i \in \mathsf{vars}(c')$.

Beweis:
1.Fall: $c.post(s) \in \mathsf{instr}(P)$: Dann gilt wegen Kor. 3.4.1 in jedem Fall $d_i = d'_i$, denn $P = P'$. Also ist die Implikation erfüllt, weil die Prämisse nicht gilt.

2.Fall: $c.post(s) \notin \mathsf{instr}(P)$:

1. Die Eigenschaft d_i wurde nicht verändert, also $d_i = d'_i$:
   dann gilt die Implikation $d_i \neq d'_i \Rightarrow v_i \in \mathsf{vars}(\mathsf{betr}(c))$, weil die Prämisse nicht erfüllt ist.

2. Die Eigenschaft ist in $\mathsf{res}(P')$ anders als in $\mathsf{res}(P)$, also $d_i \neq d'_i$:

   (a) c oder $c' \in \mathsf{builtin}(c)$ verändert d_i, dann gilt wegen Def. 3.5.3 $c \in \mathsf{betr}(c)$ und somit $\exists c'' \in \mathsf{betr}(c) \cup \mathsf{builtin}(c) : v_i \in \mathsf{vars}(c'')$.

   (b) es existiert eine Anweisung $c'.post(s') \in \mathsf{instr}(P)$ so daß c' die Variablendomäne d_i verändert (und somit $v_i \in \mathsf{vars}(c')$ gilt), dann muß einer der Fälle i., ii. oder iii. zutreffen, weil Constraints in den Variablen als Sequenzen gespeichert werden (Def. 2.1.2 Punkt 2):

      i. $c'.post(s')$ wird **vor** $c.post(s)$ in P ausgeführt:
      Dann muß c' wegen Kor. 3.4.1 während der Propagation von c erneut aufgerufen worden sein und somit gilt wegen Def. 3.3.5.6, bzw. Abb. 3.2, Zeile 7 und wegen Def. 3.5.1, daß $c' \in \mathsf{cons}(c)$, und damit wegen Def. 3.5.3 auch $c' \in \mathsf{betr}(c)$.

      ii. $c'.post(s')$ wird **nach** $c.post(s)$ in P ausgeführt:
      Dann muß c' wegen Kor. 3.4.1 die Domäne von v_i wegen einer Variable $v' \in \varepsilon(\mathsf{vars}(c'))$ eingeschränkt haben, deren Domäne direkt oder indirekt von c eingeschränkt worden ist. Damit muß

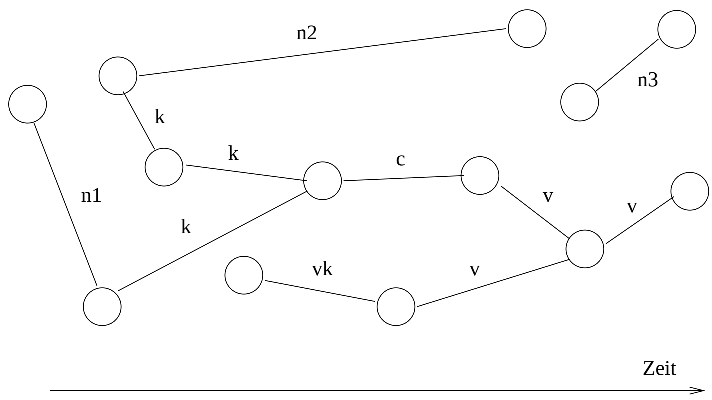

Abbildung 3.8: Die betroffenen Constraints eines Constraints c. Mit k sind die Konsequenzen, bzw. mit v die verwendenden Constraints von c markiert. Das Constraint vk ist die Konsequenz eines verwendenden Constraints. Die anderen Constraints sind nicht betroffen: n1 weil es bei der Propagation nicht nochmal abgesetzt wurde, n2, weil es nach c abgesetzt wurde und kein verwendendes Constraint ist, n3, weil es mit c im Constraintnetz nicht verbunden ist

ein "chronologischer Pfad" im Constraintnetzwerk von c nach c' existieren, so daß $c' \in \mathsf{verw}(c)$ und somit auch $c' \in \mathsf{betr}(c)$ gilt. Der "chronologische Pfad" ist eine Sequenz von Constraints, deren Elemente und Reihenfolge sich aus den Sequenzen der abgespeicherten abhängiogen Constraints ableiten lässt. wie es in Def. 3.5.2 bzw. Abb. 3.7 dargestellt wird.

iii. $c'.post(s')$ wird **vor und nach** $c.post(s)$ in P ausgeführt: Im Unterschied zu i. sei $c' \notin \mathsf{cons}(c)$, dann muß wegen Kor. 3.4.1 c' wegen eines $c'' \in \mathsf{verw}(c)$ erneut ausgeführt worden sein. Damit gilt aber $c' \in \mathsf{cons}(c'')$ also auch $c' \in \mathsf{cons}(\mathsf{verw}(c))$ und somit auch $c' \in \mathsf{betr}(c)$.

□

Um ein Constraint zurückzunehmen, um also all seine Auswirkungen auf das Constraintnetz zu revidieren, werden die betroffenen Constraints und ihre Variablen betrachtet. Zuerst werden die Domänen der betroffenen Variablen so weit vergrößert, wie es sich aus den gespeicherten Domänenurbildern (Def. 3.2.5 Punkt 6 bzw. Abb. 3.4, Zeile 4) ableiten läßt. Im zweiten Schritt werden dann alle betroffenen Constraints nochmals abgesetzt, um bei ihrer Ausführung die erweiterten Domänen wie gewünscht einzuschränken.

Definition 3.5.4 (ACS Constraintrücknahmemethode) Die *ACS Constraintrücknahmemethode* eines Constraints c in einem Constraintprogramm P überführt den Zustand $(\langle D_1, ..., D_n\rangle, \{A_1, ..., A_m\})$ von P in den Zustand $(\langle D'_1, ..., D'_n\rangle, \{A'_1, ..., A'_m\})$, wobei gilt:

1. Domänenerweiterung: $\forall i \in 1..n : D'_i := \bigcup_{c' \in \mathsf{betr}(c)} \mathsf{dom}(h_j)$
   für $\mathsf{dom}(v_j) = D_i$ und $c' = ((v_1, ..., v_j, ..., v_l), T, s, z, E, (h_1, ..., h_j, ..., h_l))$

2. Neuberechnung: $\forall c' = (\overline{v}, T, (z_i, A_i, \rho_i), zust, E, \overline{h}) \in \mathsf{betr}(c) \setminus \{c\} : A_i' := \mathsf{insert}(c', T, A_i)$

Bemerkung 3.5.3
Die ACS Constraintrücknahmemethode erzeugt i.d.R. keinen Zustand, der vorher schon bei der Ausführung des Programms vorgelegen hat. Es werden lediglich die Auswirkungen des zurückgenommenen Constraints beseitigt und die Constraints nochmals abgesetzt, die zur Herstellung des korrekten Ergebnisses durch Propagation notwendig sind.

Neben der Neuberechnung der Variablendomänen muß beim Zurücknehmen eines Constraints c auch ein Systemzustand hergestellt werden, in dem c keine Auswirkungen mehr hat. Insbesondere muß also verhindert werden, daß c bei späteren Berechnungen berücksichtigt wird und daß c aus dem Speicher entfernt werden kann. Dazu müssen beim ACS die Referenzen auf c aus den Variablen und aus den direkten Konsequenzen anderer Constraints entfernt werden. Diese Aufgaben werden im ACS-*retraction*-Algorithmus in Abb. 3.9 neben der Ausführung einer Constraintrücknahmemethode (Def.3.5.4) durchgeführt.

```
foreach c' ∈ builtin(c) do c'.retract()                       // alle Hilfsconstraints löschen
end
c = (v̄, T, s, inactive, E, h̄);                                 //c als ungültig markieren
foreach c' = (v̄', T', acs', s', E', (h_1, ..., h_n))) ∈ betr(c) do
                                                               // in allen betroffenen Constraints...
  foreach v_i = (p̄, C) ∈ ε(v̄') do                              // ...und allen Variablen
    C' := C \ {c}                                              // entferne c aus Propagationslisten
    v_i = (dom^{-1}(dom(h_i) ∪ dom(p̄)), C')                     // erweitere Domäne
  end
  E' = E' \ {c}                                                // entferne c aus Konsequenzen
end
foreach c' = (v̄', T', acs', z', E', h̄') ∈ betr(c) \ {c} do
  if z' = active                                               // alle gültigen Constraints...
    then c'.post(acs');                                        // ...nochmal absetzen
  fi
end
```

Abbildung 3.9: Ausführung der Constraint-*retraction* $c.retract()$ mit der Methode $remove()$ (Abb. 3.4) eines Constraints c.

Zu zeigen bleibt, daß der Algorithmus zur Zurücknahme von Constraints in ACS (Abb. 3.9) terminiert. Dies ist genau dann der Fall, wenn die Neuberechnung der Variablendomänen terminiert, was in Abschnitt 3.4 mit den Lemmata 3.4.3 und 3.4.4 untersucht wurde. Die Akkumulation der betroffenen Constraints und die Domänenerweiterung terminiert in jedem Fall.

Lemma 3.5.4 Wenn ein Constraintprogramm P ein Ergebnis hat, dann terminiert die Ausführung der ACS Constraintrücknahmemethode für jedes $c \in \mathsf{constraints}(P)$.

Beweis:
Wegen Lemma 3.5.1 und Lemma 3.5.2 ist die Menge $\mathsf{betr}(c) \subseteq \mathsf{constraints}(P)$ aus Def. 3.5.4 endlich. Damit ist für jedes $v_i \in \mathsf{var}(P)$ die Vereinigung $D'_i := \bigcup_{(\overline{v},T,acs,z,E,(h_1,\ldots,h_n))\in\mathsf{betr}(c)} \mathsf{dom}(h_i)$ in Def. 3.5.4.2 endlich. Weil laut Bem. 3.3.1 auch die Menge $\mathsf{var}(P)$ endlich ist, terminiert die Domänenerweiterung (Def. 3.5.4.1).

Da laut Voraussetzung P ein Ergebnis hat, also terminiert, hat laut Lem. 3.4.4 auch die Neuberechnung (nach Def. 3.5.4.2) ein Ergebnis und terminiert somit ebenfalls.

□

Fallstudie, Teil 10
Neben der Verwendung in Suchalgorithmen (vgl. Kapitel 5) wird Constraintretraction in $\mathcal{M}$Plan auch verwendet, um Termine abzusagen oder zu verschieben. Soll ein Termin abgesagt werden, so müssen die Constraintvariablen, die ihn repräsentieren, gelöscht werden. Dazu müssen alle Constraints über ihnen zurückgenommen werden, denn diese brauchen gültige Referenzen zu ihren Variablen, damit die Propagationsmethoden definiert sind. Außerdem sollen die Einschränkungen, die sich aus dem abgesagten Termin ergeben, rückgängig gemacht werden, so daß in der reservierten Zeit ein anderer Termin stattfinden kann.

3.5.4 Korrektheit

Im letzten Abschnitt wurde die Terminierung der ACS Constraintrücknahme gezeigt. Damit hat ein Constraintprogramm mit Constraint-Rücknahmen (Def. 3.3.1.5) genau dann ein Ergebnis, wenn alle abgesetzten Constraints terminieren. Jetzt zeige ich, daß das berechnete Ergebnis genau das ist, das berechnet worden wäre, wenn das zurückgenommene Constraint nie abgesetzt worden wäre.

Lemma 3.5.5 Gegeben seien die Constraintprogramme P_1 und P_2, so daß $\mathsf{instr}(P_1) = \mathsf{instr}(P_2) \cup \{c.post(s), c.retract()\} \wedge \{c.post(s), c.retract()\} \cap \mathsf{instr}(P_2) = \emptyset$ gilt und $c.post(s)$ in P_1 vor $c.retract()$ ausgeführt wird.
Wenn $P_2 \cup \{c.post(s)\}$ ein Ergebnis hat dann gilt $\mathsf{res}(P_1) = \mathsf{res}(P_2)$.

Beweis:
Wegen Lemma 3.5.4 und weil $P_2 \cup \{c.post(s)\}$ laut Voraussetzung terminiert, terminiert auch P_1, so daß $\mathsf{res}(P_1)$ existiert.

Das Lemma wird zunächst unter der Annahme bewiesen, daß P_2 additiv ist, d.h. $\neg\exists$`c'.retract()` $\in \mathsf{instr}(P_2)$. Ohne Beschränkung der Allgemeinheit wird außerdem angenommen, daß die von P_1 und P_2 generierten Constraintschemata gleich sind. Dies ist keine Beschränkung, denn laut Voraussetzung sind alle Variablen betreffenden Anweisungen in beiden Programmen gleich. Daraus ergibt sich ebenfalls, daß die initialen Variablendomänen (Def. 3.3.1.2) in beiden Programmen gleich definiert werden.

Wegen Lemma 3.5.3 genügt es zu zeigen, daß die Domänen aller $v \in \mathsf{vars}(c')$ mit $c' \in \mathsf{betr}(c)$ nach der Programmausführung gleich sind:

Wegen Def. 3.5.4.1 gilt

$$\forall v \in \mathsf{vars}(c'), c' \in \mathsf{betr}(c) : \mathsf{dom}(v) = \bigcup_{(\overline{v},T,acs,z,E,(h_1,\ldots,h_n)) \in \mathsf{betr}(c)} \mathsf{dom}(h_i) \quad (3.3)$$

Weiterhin gilt wegen Def. 3.5.4.2

$$\forall c' = (\overline{v}, T, s, z, E, \overline{h}) \in \mathsf{betr}(c) \setminus \{c\} : \exists (z, A, \rho) \in \mathsf{solvers}(P_1) : \\ \forall (f, w) \in T : post(c', w) \in \varepsilon(A) \quad (3.4)$$

Wegen der Annahme, daß P_2 additiv ist, folgt aus (3.3) und (3.4) wegen Korollar 3.4.1, daß $\mathsf{res}(P_1) \downarrow \mathsf{vars}(c') = \mathsf{res}(P_2) \downarrow \mathsf{vars}(c')$ für $c' \in \mathsf{betr}(c)$ gilt. Wegen Lemma 3.5.3 gilt damit aber auch

$$P_2 \text{ additiv } \Rightarrow \mathsf{res}(P_1) = \mathsf{res}(P_2) \quad (3.5)$$

Nun kann durch Induktion über die Anzahl der Constraintrücknahmen gezeigt werden, daß das Lemma auch gilt, wenn P_2 nicht additiv ist.
Sei n die Anzahl der Anweisungen der Form $c'.retract()$ in einem Constraintprogramm P

I.A.: $n = 0$: $\mathsf{res}(P) = \mathsf{res}(P')$ gilt falls $\mathsf{instr}(P) = \mathsf{instr}(P') \cup \{c.post(s), c.retract()\}$
Beweis: Aussage (3.5)

I.V.: $\mathsf{res}(P) = \mathsf{res}(P')$ gilt falls $\mathsf{instr}(P) = \mathsf{instr}(P') \cup \{c.post(s), c.retract()\}$ und $n > 0$.

I.S.: $\mathsf{res}(P) = \mathsf{res}(P')$,
falls $c'.retract() \in \mathsf{instr}(P) \wedge \{c'.retract(), c.post(s), c.retract()\} \subseteq \mathsf{instr}(P')$:

1.Fall: $\neg\exists c'.post(s') \in \mathsf{instr}(P)$:

$$\overset{Def.3.3.4.3}{\Rightarrow}$$

$$c' = (\overline{v}, T, null, inactive, \emptyset, null)$$

$$\overset{Def.3.5.1, Def.3.5.2}{\Rightarrow}$$

$$\mathsf{cons}(c') = \emptyset \wedge \mathsf{verw}(c') = \emptyset$$

$$\Rightarrow$$

$$\mathsf{cons}(c') \cup \mathsf{cons}(\mathsf{verw}(c')) \cup \mathsf{verw}(c') = \emptyset$$

$$\overset{Def.3.5.4, Def.3.4.3}{\Rightarrow}$$

$$\mathsf{res}(P') = \mathsf{res}(P)$$

2.Fall: $\exists c'.post(s') \in \mathsf{instr}(P)$:

$$\Rightarrow$$

$$\mathsf{instr}(P') = \mathsf{instr}(P) \cup \{c'.post(s'), c'.retract(), c.post(s'), c.retract()\}$$

$$\overset{2 \text{ mal } I.V.}{\Rightarrow}$$

$$\mathsf{res}(P) = \mathsf{res}(P')$$

□

Korollar 3.5.1 Wenn die letzte Anweisung bzgl. eines Constraints c in einem Constraintprogramms P die Rücknahme von c, also $c.retract()$, ist, dann ist das Ergebnis von P das gleiche, wie wenn c in P niemals abgesetzt worden wäre.

Bemerkung 3.5.4
Für das Ergebnis eines Constraintprogramms ist es also irrelevant, wie oft ein Constraint c abgesetzt und zurückgenommen wurde, entscheidend ist, ob die letzte Anweisung $c.post()$ oder $c.retract()$ war.

3.6 Anwendbarkeit von ACS

Zusammenfassung. Die Korrektheit und Vollständigkeit von Constraintprogrammen, in denen Constraints beliebig abgesetzt und zurückgenommen werden, kann nun zusammenfassend gezeigt werden. Damit kann man Asynchrone Constraintlöser zum Lösen dynamischer CSPs verwenden. Die Möglichkeiten zur Constraintverarbeitung mit ACS werden nach den Leistungsmerkmalen aus Abschnitt 3.1 kurz evaluiert.

Es kann nun gezeigt werden, daß das ACS Ausführungsmodell zum inkrementellen Lösen dynamischer Constraintprobleme geeignet ist. Als Ergebnis der Ausführung additiver Constraintprogramme habe ich in Abschnitt 3.4 den Fixpunkt der Propagation aller abgesetzten Constraints identifiziert. Dieser Fixpunkt kann theoretisch eindeutig aus den Implementierungen der abgesetzten Constraints abgeleitet und mittels chaotischer Iteration berechnet werden. In Abschnitt 3.5 habe ich gezeigt, daß Constraintprogramme mit Constraint-Rücknahme ebenfalls das gewünschte Ergebnis liefern. Das Zurücknehmen von Constraints verändert nicht das Ergebnis der Propagation. Diese Ergebnisse fasse ich in folgendendem Satz zusammen.

Satz 2 Sei ein Constraintprogramm P mit dem generierten Constraintschema $\langle v_1, ..., v_n\rangle$ gegeben. Wenn P ein Ergebnis hat, dann gilt $\mathsf{res}(P) = \langle D_1, ..., D_n\rangle \Leftrightarrow \mathsf{sem}(P) = D_1 \times ... \times D_n$.

Beweis:
Wegen Lemma 3.4.5 existiert $\mathsf{sem}(P)$, weil P ein Ergebnis hat. Nach Bemerkung 3.3.2 und Lemma 3.4.2 werden von P genau die Propagationsmethoden induziert, die in Constraints definiert wurden, die nicht zurückgenommen wurden. Das Ergebnis von P ist also das gleiche, wie das von

$$\begin{aligned} P' = \langle & c_1 = \texttt{new Constraint } (\overline{v}_1, T_1);\ c_1.\texttt{post}(s_1); \\ & c_2 = \texttt{new Constraint } (\overline{v}_2, T_2);\ c_2.\texttt{post}(s_2); \\ & ...;\ c_n = \texttt{new Constraint } (\overline{v}_n, T_n);\ c_n.\texttt{post}(s_n)\rangle \end{aligned}$$

mit $\forall i \in 1..n : c_i \in \mathsf{constraints}(P)$
Damit folgt der Satz direkt aus den Lemmata 3.4.6 und 3.4.7 sowie Korollar 3.5.1.

□

ACS ist ein Ausführungsmodell für unvollständige Constraintlöser (Def. 2.2.3), das mit Propagation Lösungen von CSPs finden kann. Im nächsten Kapitel wird dann beschrieben, wie man ACS durch Verwendung von Suchalgorithmen zu einem Ausführungsmodell für vollständige Constraintlöser erweitern kann. Eine ACS-Implementierung wie z.B. J.CP hat folgende, für die Constraintverarbeitung relevanten Eigenschaften, die sich aus den Definitionen dieses Kapitels ergeben:

- **Konsistenz** kann in den Propagationsmethoden jederzeit geprüft werden (Def. 3.3.5.4). Da diese Methoden immer dann ausgeführt werden, wenn sich eine Änderung einer Variablendomäne ergibt, (Def. 3.3.6.6) kann damit auch jede Inkonsistenz festgestellt und verarbeitet (Abb. 3.3, Zeilen 2-4) werden.

- **Determination**, also das Feststellen, daß eine Variable nur noch genau einen möglichen Wert hat, kann in den Variablen bei der Ausführung der Domänendefinition (Abb. 3.2) implementiert werden. Wenn eine Variable nur noch einen möglichen Wert hat, so werden alle Constraints, die von dieser Eigenschaft abhängen (Def. 2.1.2 Punkt 2) erneut ausgeführt (*forward-checking*).

- **Effizienz** wird beim ACS durch die Verwaltung der Aufgabenpuffer in den Constraintlösern (Def. 3.2.2) optimiert. Es wird also nicht nur wie in der CLP oder bei Mozart der Suchraum durch Propagation verkleinert (Abschnitt 2.2), sondern die Herstellung der Konsistenz (Def.2.2.2) wird effektiver organisiert und bringt so weitere Gewinne. Das wird in Abschnitt 7.1 noch genauer beschrieben.

- **Inkrementalität** ist in den Operationen zum Absetzen und zum Zurücknehmen von Constraints beim ACS gegeben. Der Informationsgewinn, der sich durch das Absetzen eines Constraints ergibt, wird (verzögert, durch die asynchrone Ausführung) direkt im Constraintnetz verbreitet (Abschnitt 3.4). Auch das Zurücknehmen von Constraints wird in einem bestehenden System ausgeführt, ohne alle Berechnungen erneut durchzuführen (Def. 3.5.2 und Abb. 3.9). In beiden Operationen müssen keine neue Berechnung aller Variablendomänen durchgeführt werden, sondern nur diejenigen Domänen angepaßt werden, die durch die neue Information betroffen sind.

- **Adaptivität** ist durch die Möglichkeit, Constraints zurückzunehmen (Abschnitt 3.5), gegeben.

- **Asynchronität** ist durch die Pufferung der Aufgaben (Def. 3.3.4.4 und 5) gegeben. Durch Synchronisation der Ein-und Ausgabe in den Puffern können so mit ACS auch nebenläufige Constraintprogramme bearbeitet werden. Es ist dabei nicht notwendig, eine Reihenfolge einzuhalten, weil im Unterschied zu den meisten etablierten Systemen keine zeitbezogene Verwaltung der Historie notwendig ist.

- **Erweiterbarkeit** Neue Constraints können durch Implementierung von Propagationsmethoden in bestehende Systeme integriert werden. Neue Constraintdomänen lassen sich durch neue Domänenrepräsentationen integrieren. Dies wird in Kapitel 4 genauer beschrieben.

- **Kapselung** von Constraints ist durch ihre Implementierung als Objekte gegeben. Die Kapselung von Suchalgorithmen, die sich z.B. in Mozart, ILOG und JCL bewährt hat, wird durch die Implementierung von Suchalgorithmen als Constraints (Kapitel 5) auch in ACS umgesetzt.

Kapitel 4

Die Implementierung J.CP

In diesem Kapitel wird ein kurzer Überblick über die aktuelle Implementierung des ACS Ausführungsmodells gegeben. Das System J.CP besteht zur Zeit aus drei Java-Paketen, mit denen sich einige klassische CSPs lösen lassen. Es werden die Architektur des Systems und einige Anwendungsbeispiele angegeben. Die Implementierung läßt sich leicht um neue Constraints erweitern, indem das dafür vorgesehene *interface* implementiert wird. Die sonst sehr komplexe Verarbeitung von heterogenen Constraints, also Constraints über Variablen aus unterschiedlichen Constraintdomänen, ist durch die Typisierung von Java und die Architektur von J.CP kein besonderes Problem. Heterogene Constraints und neue Constraintsysteme lassen sich direkt in das System integrieren, was an einigen Beispielen erläutert wird.

Das ACS Ausführungsmodell wurde als Java-Paket mit zwei weiteren Paketen für vordefinierte Constraintsysteme implementiert. Die unterstützten Constraintsysteme dieser prototypischen Implementierung sind endliche Domänen ganzer Zahlen [Hen89, CD96] und endliche Domänen symbolischer Werte, die explizit als beliebige Java-Objekte aufgezählt werden, wie z.B. in JCL [JCL]. Für beide Systeme sind einige grundlegende Constraints implementiert, die in den üblichen *benchmark*-Programmen verwendet werden. Weitere Constraints und Constraintsysteme lassen sich als Implementierungen bestimmter Java-*interfaces* leicht integrieren, was ich in Kapitel 4.1 genauer beschreiben werde. Es lassen sich damit auch heterogene Constraints [BS98, Hof98, RS00b] wie z.B. Labeling-Constraints über Variablen unterschiedlicher Constraintsysteme, implementieren. Zur Suche stehen zwei generische, d.h. Constraintsystem-unabhängige, Constraints zur Verfügung, die in Abschnitt 5.4 beschrieben werden.

Ich habe mich für Java als Implementierungssprache entschieden, weil für diese Sprache sehr viele APIs zur Implementierung fast aller Stand-der-Technik Methoden zur Verfügung stehen. In J.CP mache ich z.B. Gebrauch des `java.rmi` Paketes für verteilte Anwendungen oder der `Thread` und `Event` Verarbeitung, die mittlerweile zum Sprachumfang von jdk [Java] gehören. Constraintprogrammierung in Java ist ein relativ neuer Ansatz, für den sich noch keine eindeutige Syntax durchgesetzt hat. Durch die Verwendung dieser Sprache ergeben sich keine Einschränkungen sondern neue Freiräume im Vergleich zur logischen Programmierung, die in dieser Arbeit an vielen Stellen genutzt werden. Um trotzdem über Details der Constraintprogramme anhand des Quellcodes reden zu können versuchen wir, eine einheitliche Syntax zu etablieren [SR02, pooc], mit der constraintverarbeitende Systeme von Java aus angesprochen werden können.

In Abb. 4.1 wird prototypisch ein Klassendiagramm für eine J.CP Anwendung dargestellt. Der Solver läuft in einem nebenläufigen *thread*, der mit der Erzeugung des Solver-Objekts gestartet wird. Dieser *thread* ruft dann Propagationsmethoden, deren Referenzen er aus dem Aufgabenpuffer des Solvers bekommt, auf. Der Aufgabenpuffer kann auf unterschiedliche Weise implementiert werden (die Laufzeit wird in Abschnitt 7.1 untersucht). Die Zuverlässigkeitswerte, mit denen der Solver dann immer den aktuellen Fortschritt der Propagation repräsentiert, werden aus den Aufgaben im Puffer und deren Prioritäten berechnet. Bei einer entsprechenden Berechnung der Zuverlässigkeit kann jede Art von Konsistenz (Def. 2.2.2) aus dem Erreichen eines bestimmten Wertes abgeleitet werden. Die Puffer beachten dabei auch Determination, d.h. wenn n Variablen eines m-ären Constraints instantiiert sind, so wird das Constraint im Puffer behandelt wie ein $(m-n)$-äres Constraint, also früher ausgeführt.

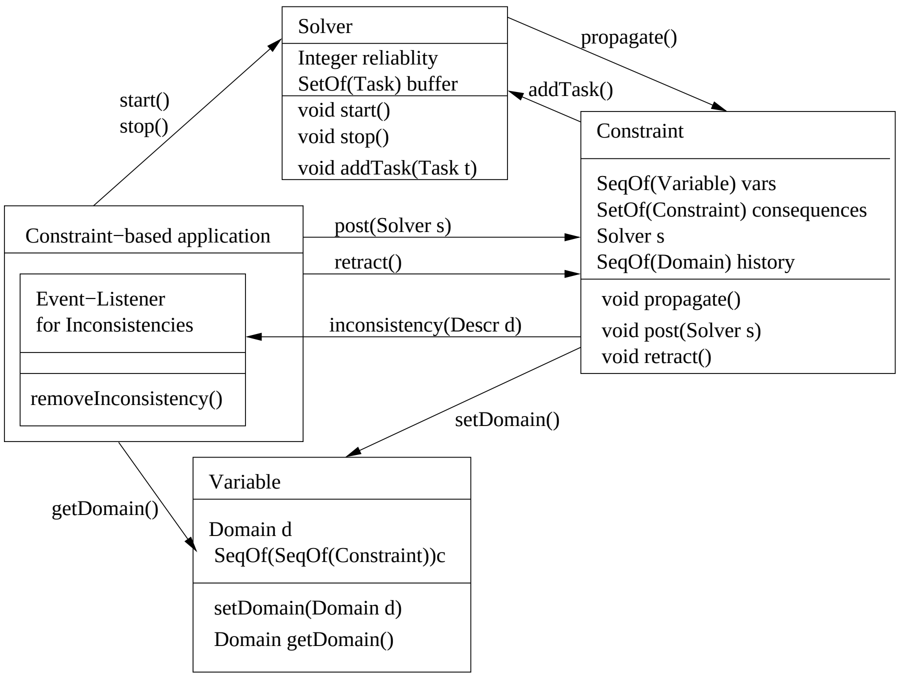

Abbildung 4.1: Architektur einer Constraintbasierten Anwendung mit ACS.

Die Behandlung von Inkonsistenzen durch Reparaturoperationen wird in J.CP durch das Java-Event-Modell umgesetzt. Wenn ein Constraint oder eine Variable eine Inkonsistenz entdeckt, so wird ein entsprechendes Ereignis erzeugt, das von sogenannten *EventListener*-Objekten erkannt wird. Diese rufen dann eine bestimmte Methode als Reaktion auf das Ereignis auf. Diese Methode (`removeInconsistecy()` in Abb. 4.1) ist normalerweise eine Reparaturoperation im Sinne von Def. 3.2.3, die also die Inkonsistenz beseitigt. Treten Inkonsistenzen bei der Propagation der Constraints auf, die durch das Programm abgesetzt wurden, so ist normalerweise unklar, wie reagiert werden soll. Daher sollte bei der Erzeugung eines Solver-Objekts immer ein solcher Event-Listener angegeben werden. Die Inkonsistenzen, die durch falsche Entscheidungen in nichtdeterministischen Suchalgorithmen auftreten, werden durch die Suchconstraints selbst beseitigt.

Fallstudie, Teil 11
Das Anwendungsprogramm $\mathcal{M}$Plan verwendet den Solver J.CP. Jeder Agent hat einen eigenen Solver, in dem alle Constraints des lokalen Agenten abgesetzt werden. Wie in Abb. 3.1 dargestellt, verwendet jeder Agent außerdem eine Datenbank für die persönlichen Präferenzen und Daten des Benutzers, eine Kommunikationsschnittstelle für andere Agenten und eine Benutzeroberfläche. Der `InconsistencyEventListener` *von $\mathcal{M}$Plan beseitigt Inkonsistenzen durch Verschiebung von Terminen auf eine andere Zeit oder einen anderen Tag. Dazu werden die Wichtigkeiten der Termine, die der Benutzer eingeben muß, verwendet, um jeweils durch Verschieben von "unwichtigen" Terminen Zeit für Wichtiges zu schaffen.* `Appointment`*-Objekte (vgl. Teil 6 der Fallstudie) werden zu einem angegebenen Zeitpunkt oder spätestens einen Tag vor ihrem ersten möglichen Termin festgelegt, d.h. ihre Constraintvariablen werden instantiiert. Dadurch können sich Inkonsistenzen ergeben, die aber nur durch Veränderungen an den späteren Terminen Auswirkungen haben können.*

4.1 Erweiterbarkeit um Constraints

Zusammenfassung. In diesem Abschnitt wird angegeben, wie neue Constraints in J.CP integriert werden können und wie ihr gewünschtes Verhalten implementiert wird. Die Methodennamen und Constraintsystem-unabhängige Algorithmen sind in einer abstrakten Prototyp-Klasse deklariert, die von jedem anderen J.CP-Constraint implementiert wird. In den Constraints selbst wird der Propagationsalgorithmus definiert, der Variablendomänen einschränkt und Inkonsistenzen erzeugt.

Neue Constraints können in J.CP durch die Implementierung entsprechender Java-Klassen integriert werden. Um die Benutzung der Constraints im System einheitlich zu gestalten, ist im Hauptpaket `jcp` eine Prototyp-Klasse enthalten, die von jedem Constraint implementiert wird. In dieser Klasse sind abstrakte Methoden deklariert, die dann von den jeweiligen Instanzen implementiert werden müssen. Dadurch ist gesichert, daß die Namenskonventionen bei Erweiterungen eingehalten werden. Die Constraints selbst sollten dann in den jeweiligen Paketen zu den verwendeten Constraintsystemen implementiert werden. Im Paket für endliche Domänen ganzer Zahlen `jcp.fd` ist zum Beispiel eine Klasse enthalten, die das $\leq$-Constraint zwischen Variablen dieser Domäne implementiert.

Außer den abstrakten Methodennamen sind auch einige Algorithmen in der abstrakten Constraint-Klasse implementiert. Das Einfügen von Arbeitsaufgaben in den Aufgabenpuffer des Constraints zum Beispiel kann unabhängig vom jeweiligen Constraintsystem durchgeführt werden und ist somit in der Prototypklasse implementiert. Auch der Algorithmus zur Constraintrücknahme (Abschnitt 3.5) ist unabhängig vom Constraintsystem und somit in der abstrakten Klasse definiert. Wenn ein Constraint diese Methode anders durchführen soll, so kann sie in Java auch die geerbte Methode überschreiben.

Um ein neues Constraint zu implementieren, muß außer dem Konstruktor noch der Propagationsalgorithmus des Constraints selbst implementiert werden. Dieser kann beliebige Parameter als Eingabe verwenden, weil hier lediglich ein *array* von Objekten übergeben wird. In der Regel sind die Parameter Constraintvariablen des jeweiligen Constraintsystems, es sind aber auch beliebige andere Objekte denkbar, um weitere Informationen in den Constraints zu verarbeiten. In $\mathcal{M}$Plan wird z.B. ein Constraint verwendet, das für die Aktualisierung der Benutzeroberfläche zuständig ist, wenn ein Termin festgelegt wurde. Dieses Constraint verwendet als Parameter eine Referenz auf die Benutzeroberfläche (die keine Variable ist) und wird aktiv, sobald seine Variablen instantiiert sind, wenn es also einen Termin graphisch darzustellen gilt.

Im Propagationsalgorithmus der Constraints werden die Domäneneinschränkungen für die Variablen durch entsprechende Methodenaufrufe oder Nachrichten implementiert. Wie die Domänen eingeschränkt werden, ergibt sich aus dem jeweils implementierten Algorithmus. Außerdem können im Propagationsalgorithmus Inkonsistenzen erzeugt werden, die dann als Ereignis im System verarbeitet werden. Wenn in einem Propagationsalgorithmus festgestellt wird, daß das Constraint mit den aktuellen Variablendomänen nicht erfüllbar ist, so wird diese Information an den Solver und damit an alle bekannten `InconsistencyEventListener` weitergegeben. Abweichend von der konzeptionellen Definition 3.3.6.6 werden die Variablen in J.CP explizit von den Constraints informiert, aufgrund welcher Veränderungen ihrer Domäne die Constraints erneut ausgeführt werden sollen. In der genannten Definition werden Constraints immer dann in die Liste der abhängigen Constraints einer Domäneneigenschaft eingetragen, wenn sie diese verändert haben. In J.CP kann dies frei gestaltet werden, sollte aber mindestens die genannten Fälle abdecken.

Durch die Implementierung mehrerer Propagationmethoden mit unterschiedlichen Prioritäten kann der Effekt der Propagation eines Constraints in mehreren Stufen definiert werden. Einerseits sollte man die Propagation immer möglichst schnell abschließen, um insgesamt effizienter zu arbeiten, andererseits gibt es, vor allem bei globalen Constraints, oft recht aufwendige Algorithmen, die manchmal auch gute Suchraumeinschränkungen erzeugen. Letztere Algorithmen können in J.CP mit niedriger Priorität eingesetzt werden, so daß sie nur dann ausgeführt werden, wenn der Solver keine wichtigeren Aufgaben zu erledigen hat. Insbesondere bei interaktiven Systemen, wie z.B. $\mathcal{M}$Plan, treten während der Programmausführung oft Freiräume durch langsame Benutzereingaben auf, die zur Berechnung weitgehender Propagationsalgorithmen im Solver genutzt werden können.

4.2 Kombination von Constraintsystemen

Zusammenfassung. Bei der Modellierung von Constraintproblemen können sich unterschiedliche Constraintdomänen für die verwendeten Variablen ergeben. Sogenannte heterogene Constraints, die Beziehungen über Variablen unterschiedlicher Domänen beschreiben, sind von herkömmlichen effizienten Solvern nicht zu bearbeiten, weil diese die interne Struktur der Constraintdomänen nutzen. Dieses Problem wird bisher in akademischen Systemen durch gesteuerte Kooperation von Solvern gelöst. In J.CP können heterogene Constraints unter der Ausnutzung des Typsystems von Java direkt implementiert werden, so daß der Aufwand zur Koordination entfällt. In diesem Abschnitt wird eine kurze Einführung in den Stand der Wissenschaft gegeben und dargestellt, wie sich unterschiedliche Constraintsysteme und heterogene Constraints in J.CP integrieren lassen.

Constraintprobleme beziehen sich immer auf ein Anwendungsfeld der "realen Welt", die mit der Problemdefinition durch Variablen und Constraints modelliert wird. Je nach Anwendungsgebiet unterscheiden sich dabei die Werte, die die Variablen annehmen können, also die Elemente der Variablendomänen. Die Constraints, die diese Domänen einschränken, beziehen sich demnach auch abhängig von der Modellierung auf unterschiedliche Typen von Elementen und inferieren die Domäneneinschränkungen analog zu einer intendierten Theorie. Die Welt, in der ein Constraintproblem auf diese Weise modelliert wird, bezeichnet man als Constraintsystem [FA97]. Das Constraintsystem ergibt sich in der Regel aus der Constraintdomäne (Def.2.1.1), denn der Typ der Variablen kann als Trägermenge einer Algebra angenommen werden, deren Operationen und Relationen dann in den Constraints implementiert werden.

Definition 4.2.1 (Constraintsystem) Gegeben sei eine Constraintdomäne D, dann besteht ein *Constraintsystem* für D aus einer Menge von zulässigen Constraints C, deren Implementierung ihre deklarative Semantik in D durch eine Propagationsfunktion induziert. Desweiteren ist für jedes $c \in C$ die Erfüllbarkeit entscheidbar.

Für einige bekannte Algebren gibt es sehr effiziente Constraintsysteme, die die interne Struktur der Constraintdomänen nutzen, um besonders schnell Propagation oder Suche durchführen zu können. Man kann zum Beispiel mit dem Simplex-Algorithmus [NM93] sehr effizient Constraintprobleme in der Domäne der reellen Zahlen lösen, was z.B. für clp(R)[JMSY92] oder ILOG Concert[Conc] implementiert wurde. Analog existieren effiziente Algorithmen zur Herstellung von Konsistenz oder zum Finden von Lösungen für arithmetische Constraints in den ganzen Zahlen, z.B. in clp(FD)[CD96] oder CHIP[Chip]. Diese Systeme verwenden ebenfalls spezielle Algorithmen, die die Eigenschaften der Algebra $\mathbb{Z}$ nutzen, um immense Performanzgewinne im Vergleich zur rein symbolischen Verarbeitung der Domänen, wie z.B. in JCL[JCL] zu erreichen.

Manche Constraintprobleme der "realen Welt" lassen sich nur schwer in einem dieser effizient bearbeitbaren Constraintsysteme modellieren. Um z.B. das Problem aus Abb. 4.2 durch Constraintverarbeitung zu lösen, ohne dabei ein wesentlich komplexeres und realitätsferneres Modell zu wählen, müssen Constraints unterschiedlicher Constraintsysteme verwendet werden. Solche Constraintprobleme werden als heterogene Constraintsysteme [BS95] bezeichnet.

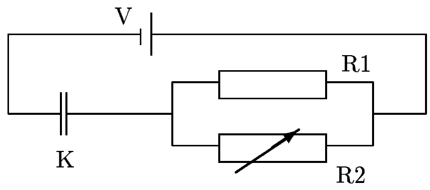

Abbildung 4.2: Der Stromkreis mit $R_1 = 0.1M\Omega, R_2 \in [0.1M\Omega;\ 0.4M\Omega]$ und zur Auswahl stehenden Kapazitäten aus der Menge $K \in \{1\mu F, 5\mu F, 20\mu F, 50\mu F\}$ kann nicht in einer bekannten Constraintdomäne modelliert werden. Das Problem ist, festzustellen, welche Kapazität verwendet werden muß, damit nach $t \in \{500..100\}$ Millisekunden 99% der Endspannung V erreicht werden. Für den Einstellwiderstand R_2 wird die Arithmetik rationaler Zahlen, für die möglichen Kapazitäten die Verarbeitung von Aufzählungen und für die Zeit t werden Intervalle ganzer Zahlen benötigt. Durch Constraints kann das Problem folgendermaßen modelliert werden, wobei die Initialisierung der Domänen vorausgesetzt wird: $\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}, V_K = V * (1 - exp(\frac{-t}{R*K})), V_K = 0.99 * V$.

4.2.1 Stand der Wissenschaft

Um die Interaktion getrennter, effizienter Solver für unterschiedliche Constraintsysteme zum Lösen heterogener Constraintprobleme zu verwenden, werden in der Literatur [Hof01, Mon98, Mon96, BS98] folgende Bearbeitungsschritte angegeben:

1. Zerlege das Problem in Teilprobleme, die von spezialisierten Solvern bearbeitet werden können.
2. Lasse diese Probleme lösen.
3. Füge die Ergebnisse zusammen.

Um die heterogenen Terme zu zerlegen, werden dabei in allen Ansätzen Variablen für nicht bearbeitbare Teilprobleme erzeugt, die als Platzhalter für das Ergebnis eines Solvers in einem anderen Solver verwendet werden. Diese Idee wird in [BS98] genau beschrieben und verifiziert. Das heterogene Constraintproblem $X - 2 < sin(Y)$ über der *finite domain* Variable $X\texttt{in}1..12$ und der reellen Variable $Y \in [1, 5]$ kann zum Beispiel durch die Konjunktion der (homogenen) Constraints $X - 2 < Z \wedge Z = sin(Y)$ äquivalent ersetzt werden. Letztere Darstellung kann durch zwei spezialisierte Solver jeweils teilweise gelöst werden, so daß die eigentliche Verbindung der Constraintsysteme durch die angepasste Interpretation der neu erzeugten Variable Z erfolgt. Für diese Variable sind die Constraints ($<$ und $=$) beider Systeme interpretierbar, so daß die Berechnung der Domäne von Z mit der Gleichung, nämlich $Z \in [-1, 1]$ in der Ungleichung verwendet werden kann. Es ergibt sich dort $X < 3$ und somit eine Einschränkung der Domäne von X.

Diese Idee wurde in einigen Ansätzen umgesetzt, wobei es meist recht aufwendig ist, die Constraints geeignet zu zerlegen. Außerdem hat es sich oft herausgestellt, daß speziell für ein solches System konzipierte Solver notwendig sind. In Solex [MR98], BALI [Mon98] oder meta-solver [HSG01, Hof01] werden die bekannten Ausführungsmodelle und effizienten Algorithmen verwendet, um spezielle Solver, die in ein Gesamtsystem zur Solver-Kooperation integrierbar sind, zu implementieren.

Neben der Kooperation der Solver ist in einem heterogenen System außerdem das Problem der Namenskonflikte zu lösen. Wenn ungetypte Sprachen verwendet werden, so kann z.B. das $<$-Constraint in den ganzen Zahlen oft nicht von dem in den reellen Zahlen unterschieden werden, ohne vorher die Typen seiner Argumente zu inferieren. Die hier entstehenden Probleme können durch herkömmliche Methoden der Typinferenz und der Strukturierung des Namensraums durch Erweiterung der Namen um ihre Constraintsysteme gelöst werden. Dies haben wir in [RS00a, RS00b] für heterogene CHR-Programme durchgeführt und implementiert.

4.2.2 Constraintsysteme in J.CP

Constraintsysteme werden in J.CP als Java-Pakete integriert, die die entsprechenden Klassen zur Implementierung der Constraints und der Variablen enthalten. In der aktuellen Version von J.CP sind die Pakete `jcp.fd` zur Bearbeitung arithmetischer Constraints über ganzen Zahlen und `jcp.enum` für Gleichheits- und Ungleichheitsconstraints über Variablen mit aufgezählten Objekten vorhanden. Im Paket `jcp` sind Prototypklassen für Constraints, Variablen und Variablendomänen enthalten, die durch Klassen der jeweiligen Constraintsysteme implementiert werden können. Durch die Verwendung dieser Interfaces zur Objekt-Deklaration können auch generische Constraints implementiert werden die gleichermaßen für Variablen unterschiedlicher Constraintdomänen arbeiten. Solche Constraints können z.B. Suchalgorithmen implementieren, was in Abschnitt 5.3.4 beschrieben wird.

Die abstrakten Prototypklassen für Variablen und Domänen enthalten analog zur Prototypklasse der Constraints die Festlegung einiger Methodennamen, die dann von den konkreten Klassen implementiert werden müssen. Im Prototyp der Variablendomänen werden z.B. Mengenoperationen wie Schnitt und Vereinigung für die Domänen deklariert, die für jede Art von Constraintdomäne implementiert werden müssen, um die Anwendbarkeit der Propagations- und *retraction*-Algorithmen zu gewährleisten. Außer diesen Namensfestlegungen enthalten die Prototypklassen für Variablen bzw. Variablendomänen auch einige Algorithmen, die Constraintsystemunabhängig verwendbar sind. Dies ist z.B. die Methode zum Eintragen von Constraints in die Liste der abhängigen Constraints in den Variablen (vgl. Def. 2.1.2).

In Abb. 4.3 sind die Spezialisierungsbeziehungen der Klassen und die jeweiligen Pakete für die aktuelle Version von J.CP dargestellt. In den Variablen des Pakets `jcp.fd` werden dort z.B. außer der Methode 'add' zum Eintragen von Constraints in die Abhängigkeitslisten auch die Methoden 'getMin' und 'getMax' dargestellt, die die (domänenspezifischen) Ganzzahlwerte aus der Domäne liefern.

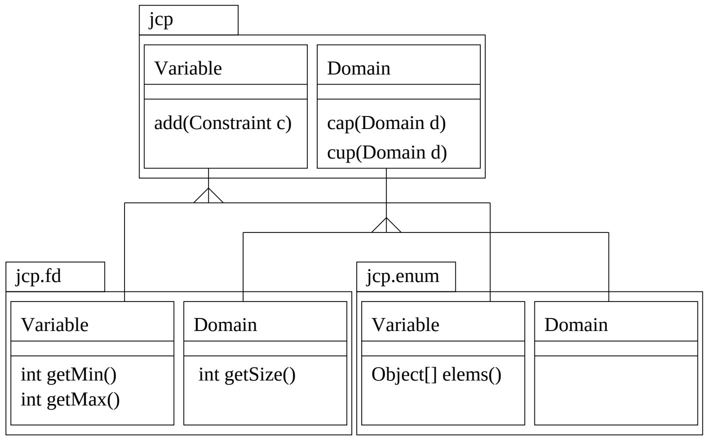

Abbildung 4.3: Spezialisierung/Generalisierung der Klassen für Variablen und Variablendomänen zur Integration der Constraintsysteme enum und fd in J.CP.

4.2.3 Heterogene Constraints in J.CP

Durch die strenge Typisierung von Java und die Möglichkeit, Constraints in J.CP relativ frei zu implementieren, können heterogene Constraints fur J.CP direkt implementiert werden. Es ist nicht nötig, unterschiedliche spezielle Solver zu kombinieren, weil die Propagationsmethoden der Constraints entsprechend implementiert werden können. Auch unterschiedliche Algorithmen zum Lösen von Constraints können in solchen heterogenen Constraints verwendet werden.

Ein heterogenes Constraint in J.CP referenziert Variablen unterschiedlicher Constraintsysteme und verwendet eine intern festgelegte Interpretation der Constraintdomäne einer Variable zur Benutzung ihrer Eigenschaften in der (anderen) Constraintdomäne einer anderen Variable. In Abb. 4.4 wird eine Implementierung der Propagationsmethode eines <-Constraints über Variablen mit ganzzahliger Domäne und Variablen mit reellwertiger Domäne dargestellt. Die Abstimmung der Constraintdomänen erfolgt hier dadurch, daß die ganzen Zahlen in den reellen enthalten sind und somit die Interpretation einer ganzen Zahl in den reellen Zahlen direkt gegeben ist. Die Interpretation der reellen Zahl in den ganzen Zahlen ist in diesem Beispiel abhängig vom Constraint. Beim $X < Y$-Constraint kann für die ganzzahlige Variable X der Absolutwert der oberen Grenze von Y angenommen werden, der mit der Funktion `int abs(float f)` berechnet werden kann. Man beachte, daß sich die Funktionalität der Methoden zum Lesen bzw. Schreiben der Grenzen der Domänen (`getMin()`,`getMax()`, `setMin()`,`setMax()`) je nach Constraintdomäne unterscheidet. In der Klasse `jcp.fd` ist z.B. `int getMin()`, in der Klasse `jcp.real` aber `float getMin()` definiert.

```
void propagate(fd.Variable x, real.Variable y){  // x < y
  if((float)x.getMin() > y.getMax()) // Vergleich mittels typecast
     broadcast(new Inconsistency(this)); // Inkonsistenz entdeckt
  else  // das Constraint ist erfüllbar
     y.setMin((float)x.getMin()); //Domänen einschränken
     int newMax = abs(y.geMax()); // der Absolutwert
     if(newMax == y.getMax()) // y.max war eine ganze Zahl
        x.setMax(newMax-1);
     else
        x.setMax(newMax);
}
```

Abbildung 4.4: Implementierung der Propagationsmethode eines heterogenen <-Constraints in J.CP.

Um unterschiedliche Ausführungsmodelle, Propagationsalgorithmen oder Suchalgorithmen für unterschiedliche Constraintdomänen in J.CP zu kombinieren, müssen diese in heterogenen Constraints implementiert werden. Es ist z.B. denkbar, einen Simplex-Algorithmus zu integrieren, der Variablen, deren Domäne irgendwie durch Zahlen interpretierbar ist, verarbeiten kann. Dazu müßte lediglich im Algorithmus immer darauf geachtet werden, daß sich die Typen der Variablen unterscheiden können. Der Simplex-Algorithus selbst könnte dann in einer Propagationsmethode eines Contraints implementiert und somit in J.CP integriert werden. Solche komplexen heterogenen Algorithmen sind momentan allerdings nur zur Suche implementiert, was im nächsten Kapitel genauer beschrieben wird.

Kapitel 5

Suche

Zur Berechnung von Lösungen für CSPs werden Suchalgorithmen in dem durch Propagation eingeschränkten Suchraum verwendet. In diesem Kapitel wird die Integration von Suchalgorithmen in das ACS Ausführungsmodell beschrieben. Die Algorithmen werden als Suchconstraints implementiert, die Instantiierungen über Variablen absetzten, deren Konsistenz durch Propagation geprüft wird. In ACS sind damit beliebige Suchalgorithmen, wie z.B. *backtracking* oder *local-search* integrierbar, deren unterschiedliche Eigenschaften bzgl. Vollständigkeit und Terminierung untersucht werden. Es werden außerdem technische Methoden angegeben, mit denen die Suchalgorithmen für die ACS Implementierung J.CP umgesetzt werden konnten.

Die Ausführung der Propagationsmethoden von Constraints in Constraint-verarbeitenden Systemen wie ACS führt im Allgemeinen nicht zu Lösungen des durch die Constraints beschriebenen CSP. Wie bereits in Abschnitt 2.2 diskutiert wurde, können durch Propagation nur manche Inkonsistenzen entdeckt werden. Es kann aber in keinem allgemeinen CSP durch die Herstellung von Konsistenz im Sinne von Def. 2.2.2 sichergestellt werden, daß das CSP wirklich lösbar ist. Analog ergeben sich aus der Berechnung der verschiedenen Arten von Konsistenz auch nur Domäneneinschränkungen der Variablen, die normalerweise noch keine Lösungen repräsentieren. Daher ist beim Constraintlösen, egal ob synchron oder asynchron, immer ein Suchalgorithmus notwendig, der konkrete Lösungen berechnet und damit die Konsistenz des CSP sichert. Weil CSPs im allgemeinen NP-vollständig sind, muß man hierzu nicht-deterministische Algorithmen verwenden, die durch Versuch und Irrtum die Suche nach Lösungen steuern. Während der Ausführung dieser Algorithmen kann jeder Versuch, also jede Variableninstantiierung auf Probe, durch Propagation auf Konsistenz überprüft werden. Daraus ergeben sich dann oft Domäneneinschränkungen anderer, noch nicht instantiierter Variablen. In zustandsbasierten Constraintlösern, wie inkrementellen CLP-Lösern [SICS, ecl], aber auch in nicht-inkrementellen Lösern [JCL, ILOG], werden in der Regel alle Instantiierungen während der Suche propagiert (*forward-checking*), so daß möglichst früh Inkonsistenzen entdeckt werden können.

In manchen Anwendungen hat sich gezeigt [CD01], daß inkrementelles *forward-checking* nicht unbedingt schneller zu Lösungen führt als lokale Veränderungen an zufällig erzeugten Variablenbelegungen ohne Propagation. Bei diesen *local-search* Ansätzen wird versucht, Inkonsistenzen durch Neu-Instantiierungen von Variablen zu beseitigen. Dazu muß jede Variablenbelegung auf Konsistenz überprüft werden und eine Bewertungsfunktion für alternative Neu-Instantiierungen ausgewertet werden, um möglichst nur "verbessernde" Veränderungen vorzunehmen. Alternative Suchalgorithmen in der Constraintverarbeitung unterscheiden sich also (unter anderem) in der Art und Weise, wie sie die Inkonsistenzen, die durch fehlerhafte Instantiierungsversuche entstanden sind, beseitigen.

Betrachtet man Suchalgorithmen als Constraints, um sie in das Ausführungsmodell eines Constraintlösers wie ACS homogen einpassen zu können, so muß ihre Semantik heißen: "Die Domänen aller Variablen haben genau einen Wert." Eine Implementierung dieser Semantik in Propagationsfunktionen erleichtert die Integration von Propagation und Suche [Rin02d, Rin02a]. Wenn nach einer erfolgreichen Suche in einem kontinuierlichen Constraintprogramm ein neues Constraint eingegeben wird, das durch die gefundene Lösung nicht erfüllt ist, so muß das Suchconstraint reaktiviert werden und eine andere Lösung suchen. Diese Art der Reaktivierung ist bei der Propagation in inkrementellen Constraintlösern konzeptionell ohnenhin vorgesehen und kann auch für die Suche verwendet werden.

Im Unterschied zu "normalen" Constraints wird die Domäneneinschränkung der Suchconstraints nicht aus domänespezifischem Wissen über die Variablen abgeleitetet. Sie beschreiben lediglich ein theoretisches Konstrukt, das zum Lösen allgemeiner CSPs notwendig ist. Daher können Suchalgorithmen in generischen Objekten implementiert werden, deren Schnittstelle zu den Variablen lediglich aus einer Wertauswahlfunktion [Hen89] besteht. Die Variablen werden mit dem Wert instantiiert, den sie selbst vorschlagen und wenn dies zu einer Inkonsistenz führt, wird diese beseitigt. Die Reihenfolge, in der die Variablen dabei ihre möglichen Werte vorschlagen, beeinlusst stark die Durchführung der Suche, so daß diese Funktion als zweite wesentliche Eingabegröße für generische Suchalgorithmen betrachtet werden kann. Um den Suchraum durch Versuch und Irrtum vollständig durchsuchen zu können, um also einen vollständigen Constraintlöser zu erhalten, muß der Suchraum endlich sein. Die Endlichkeit des Suchraums ergibt sich aus der Endlichkeit des Bildbereichs der Wertauswahlfunktionen.

5.1 Suche als Constraintverfahren

Zusammenfassung. Weil Propagation und die Herstellung von Konsistenz nicht ausreicht, um allgemeine (NP-vollständige) CSPs zu lösen, werden nichtdeterministische Suchalgorithmen in ACS integriert. Die Algorithmen werden in generischen globalen Constraints gekapselt, die Instantiierungen über ihren Variablen absetzen und zurücknehmen bis eine oder mehrere Lösungen gefunden wurden. Im Gegensatz zu den meisten Constraintverarbeitenden Systemen hat Suche in ACS also keinen gesonderten Status als spezielle Komponenten oder Teil des Ausführungsmodells. Der verwendete Algorithmus berechnet bei der Ausführung des Suchconstraints konsistente Lösungen im globalen Suchraum.

Beim Asynchronen Constraintlösen werden nicht-deterministische Suchalgorithmen durch Instantiierungen von Variablen und Prüfung der Konsistenz dieser Instantiierungen bzgl. der bekannten Constraints implementiert. Eine zentrale Idee der CP ist, daß die Werte, mit denen Variablen während der Suche probehalber instantiiert werden, explizit als Variablendomäne verwaltet werden. Ausschließlich Werte aus der bekannten Domäne werden untersucht, weil für alle anderen Werte sichergestellt ist, daß sie nicht zu einer Lösung führen. Welche dieser Werte zuerst untersucht werden sollten, um möglichst schnell eine Lösung zu finden, kann a priori nicht bestimmt werden. Hierzu wurden einige Heuristiken untersucht, die in bestimmten Anwendungsgebieten in Kombination mit bestimmten Ausführungsmodellen zur Constraintverarbeitung besonders schnell zu Lösungen führen (z.B. [Hen89, MS98, Wal96]). Einige bekannte Heuristiken werden in Abschnitt 7.3 für spezielle Anwendungen verglichen. Zur Definition von Suchconstraints für ACS beschränke ich mich nicht auf eine bestimmte Wertauswahl-Heuristik, sondern gebe eine nicht-deterministische Wertauswahlfunktion an, die in generischen Suchalgorithmen unterschiedlich implementiert werden kann.

Definition 5.1.1 (Wertauswahlfunktion) Die partielle *Wertauswahlfunktion* $\mathsf{indomain}$ liefert zu jeder Variable $v = (\overline{p}, C)$ mit $\mathsf{dom}(\overline{p}) \neq \emptyset$ die Domänenbeschreibung der Menge $\{d\}$, wobei $d \in \mathsf{dom}(\overline{p})$ beliebig gewählt wird:

$$\mathsf{indomain}((\overline{p}, C)) = \begin{cases} undefined, & \text{falls } \mathsf{dom}(\overline{p}) = \emptyset \\ \mathsf{dom}^{-1}(\{d\}) \text{ mit } d \in \mathsf{dom}(\overline{p}), & \text{sonst} \end{cases}$$

Wenn ein Wert aus einer Variablendomäne ausgewählt wurde, so kann die Variable durch eine Instantiierung an diesen Wert gebunden werden. Diese Bindung ist beim ACS durch eine bestimmte Art von Constraints implementiert, die den Nicht-Determinismus der Wertauswahlfunktion für das asynchrone Ausführungsmodell umsetzt. Das Instantiierungsconstraint ist immer genau dann erfüllbar, wenn ein Wert existiert, mit dem seine (einzige) Variable instantiiert werden kann. Im erfolgreichen Fall wird die Domäne dieser Variable auf den ausgewählten Wert eingeschränkt. Bei der Ausführung im ACS wird diese Einschränkung dann propagiert, d.h. *forward-checking* wird asynchron ausgeführt.

Definition 5.1.2 (Instantiierung) Eine *Instantiierung* ist ein ACS Constraint der Form $inst = ((v), \{(\mathsf{indomain}_v, w)\}, acs, E, \overline{h})$, wobei für eine beliebige Wertauswahlfunktion $\mathsf{indomain}$ gilt
v.`setDomain`$(\mathsf{indomain}(v)) \in \mathsf{instr}(\mathsf{indomain}_v)$, falls $\mathsf{indomain}(v) \neq undefined$
und `broadcast(new EmptyDomainInconsistency(`v`)` $\in \mathsf{instr}(\mathsf{indomain}_v)$, sonst.
Weiterhin muß gelten, daß $\mathsf{builtin}(inst) = \emptyset$.

Die `EmptyDomainInconsistency(`v`)` aus der Definition ist dabei eine Spezialisierung des `InconsistencyEvent` aus Def. 3.3.5.4. Wenn das Ereignis eintritt, daß kein Wert mehr zur Instantiierung einer Variable in ihrer Domäne enthalten ist, so ist eine Inkonsistenz aufgetreten. Diese muß dann durch eine Reparaturoperation beseitigt werden. Wie in Abb. 3.3 dargestellt, geschieht dies sofort nachdem des Ereignis im Solver bekannt wurde.

Fallstudie, Teil 12
In $\mathcal{M}$Plan werden `Appointment`*-Objekte (vgl. Teil 6 der Fallstudie) instantiiert, indem alle Constraintvariablen, die den Termin beschreiben, instantiiert werden.*

Dazu müssen die entsprechenden Implementierungn von Variablen eine Wertauswahlfunktion indomain *zur Verfügung stellen. Das Instantiierungsconstraint hat dann folgende Propagationsmethode*

```
class AppInstantiation{
  ...
  private void runInstantiation(Appointment a){
    try{
      (new TagInstantiation(
             a.getTag(), a.getTag().indomain())).post(acs);
      (new ZeitInstantiation(
             a.getZeit(), a.getZeit().indomain())).post(acs);
      (new TagInstantiation(
             a.getDauer(), a.getDauer().indomain())).post(acs);
      (new TagInstantiation(
             a.getOrt(), a.getOrt().indomain())).post(acs);
    catch(EmptyDomainException ex){
      broadcast(new EmptyDomainInconsistency(ex,this));
    }
   ...
  }
  ...
}
```

Die Methode indomain *der verschiedenen Spezialisierungen von Constraintvariablen (*tag,zeit,dauer,ort*) liefert jeweils einen Wert aus der aktuellen Variablendomäne, der verwendet wird, um ein Instantiierungsconstraint für die jeweilige Variable abzusetzen. Ist kein solcher Wert vorhanden, so wirft* indomain *eine Ausnahme, die durch die Java-Anweisung* catch *aufgefangen wird. In diesem Fall wird ein entsprechendes Inkonsistenz-Ereignis erzeugt, das zur Auflösung der Inkonsistenz die Ausnahme selbst, die Referenz auf die kritischen Variablen durch das* AppInstantiation-*Constraint und die dort gespeicherten Daten verwenden kann.*

Um mehrere Variablen eines Constraintprogramms zu instantiieren, was normalerweise beim Lösen von CSPs notwendig ist, werden die Variablen in einem globalen Constraint "gleichzeitig" instantiiert. Solche globalen Constraints sind genau wie die zufällige Wertauswahl nur dann sinnvoll, wenn es nur endlich viele zu untersuchende Werte gibt. In Constraintsystemen, in denen Variablen unendlich viele Werte annehmen können, wie z.B. in [JMSY92], müssen andere Algorithmen angewandt werden. In dieser Arbeit beschränke ich mich aber bei der Definition von Suchalgorithmen auf endliche Suchräume und damit auf endliche Variablendomänen. Das ACS Ausführungsmodell, das im vorherigen Kapitel vorgestellt wurde, funktioniert aber auch für unendlichen Wertebereiche, bei solchen Anwendungen muß dann ein anderes Suchverfahren verwendet werden.

Definition 5.1.3 (Labeling, induzierte Variablenbelegung) Ein *Labeling* ist ein ACS Constraint c mit $\mathsf{vars}(c) = (v_1, ..., v_n)$, so daß für alle $i \in 1..n$ die Menge $\mathsf{dom}(v_i)$ endlich ist und eine Instantiierung $inst_i = ((v_i), \{(\mathsf{indomain}_{v_i}, w)\}, acs, E, \overline{h}) \in \mathsf{builtin}(c)$ existiert. Die Menge $\mathsf{builtin}(c)$ enthält dabei keine weiteren

Elemente. Das Labeling c *induziert* dann die Variablenbelegung $\mathsf{indomain}_{v_1} \circ \ldots \circ \mathsf{indomain}_{v_n}(\mathsf{dom}(v_1) \times \ldots \times \mathsf{dom}(v_n))$.

Bemerkung 5.1.1
Labeling-Constraints können, wie jedes Constraint, mehrfach ausgeführt werden indem jedesmal die Propagationsmethoden aufgerufen werden. Die Domäneneinschränkungen, die sich aus den verschiedenen Ausführungen ergeben, sind aber normalerweise unterschiedlich, weil die Wertauswahlfunktionen bei jedem Aufruf unterschiedliche Werte liefern sollten. Dadurch induziert ein Labeling bei jeder Ausführung eine andere Variablenbelegung, so daß ich im Folgenden auch von mehreren induzierten Belegungen eines Labeling sprechen werde. Dadurch, daß die Domänenreduktion von Labelings immer unterschiedlich ist, kann die Terminierung von Labeling-Constraints nicht mehr durch Lemma 3.4.3 gesichert werden, so daß ich sie im Folgenden gesondert untersuchen werde.

Die Instantiierungen aus Def. 5.1.2 sollen beim ACS ausschließlich während der Suche erzeugt und ausgeführt werden. Es sind also genau die Constraints, die die Fehlentscheidungen, die sich aus dem Nichtdeterminismus der Wertauswahl ergeben, ausgeführt haben. Während der Suche sollten also auch nur diese Constraints zurückgenommen werden, denn die Constraints des CSP, das in einem Constraintprogramm umgesetzt wurde, gelten nach wie vor und sollten daher nicht durch eine Reparaturoperation entfernt werden.

Definition 5.1.4 (Instantiierungsreparaturoperation) Eine Reparaturoperation, die ausschließlich Instantiierungen aus einem gegeben CSP entfernt, heißt *Instantiierungsreparaturoperation.*

Instantiierungsreparaturoperationen werden für ACS durch Methoden implementiert, die Instantiierungen mancher Variablen zurücknehmen.

Definition 5.1.5 (ACS Reparaturoperation) Die Implementierung einer Instantiierungsreparaturoperation für ACS in einem Constraintprogramm P hat folgende Eigenschaften:

1. Es wird eine nichtleere Menge $I \subseteq \mathsf{constraints}(P)$ von Instantiierungen ausgewählt.

2. Für jedes $inst \in I$ wird die Operation $inst.\mathtt{retract}()$ ausgeführt.

Durch ACS Reparaturoperationen, also durch das Zurücknehmen von Instantiierungen, wird ein Zustand hergestellt, der äquivalent zu dem Zustand ist der entstanden wäre, wenn die jeweiligen Instantiierungen niemals abgesetzt worden wären. Dieser Zustand hat i.d.R. vorher bei der Ausführung des Constraintprogramms niemals existiert. Hierin unterscheidet sich ACS wesentlich von anderen CP Ausführungsmodellen, wie sie in Abschnitt 2.3 beschreiben wurden. Es wird kein Zustand wiederhergestellt, der in einem gegebenen Suchraum bereits existiert hat, sondern es wird ein alternativer Suchbaum hergestellt, in dem alle nicht zurückgenommenen Entscheidungen festgehalten werden. Dazu werden die Ebenen des Baumes vertauscht. Die Ebene, die die alternativen Werte einer Variable x darstellt wird nach unten verschoben, wenn die Instantiierung von x zurückgenommen wurde. Dadurch verschieben sich die Ebenen der bereits instantiierten Variablen nach oben. Die Unterbäume werden dabei jeweils identifiziert, was dadurch möglich ist, daß die Werte entweder festgelegt wurden oder noch völlig unbestimmt sind. Um also verschiedene Instantiierungen explorativ zu untersuchen, kann man einen solchen Suchbaum nicht ebenenweise durchmustern, sondern muß bereits betrachtete Werte abspeichern, was in Abschnitt 5.3.2 genauer beschrieben wird. In Abb. 5.1 wird eine solche Umordnung eines Suchbaus exemplarisch dargestellt.

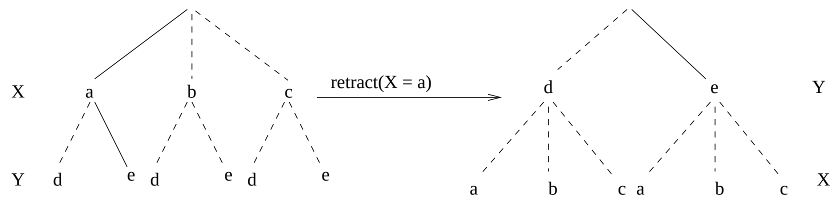

Abbildung 5.1: Umordnung des Suchbaums beim Zurücknehmen einer Instantiierung. Mit durchgezogenen Linien sind gewählte Instantiierungen markiert, gestrichelte Linien repräsentieren unbetrachtete Alternativen.

Wenn bei der Suche eine Inkonsistenz auftritt, ist i.A. nicht klar, welche Instantiierung der eigentliche Grund dafür war. Man kann also keine nicht-triviale ACS Reparaturoperation angeben, die genau die Instantiierungen zurücknimmt, die notwendig sind, um ein konsistentes Programm zu erhalten. Eine ACS Reparaturoperation kann daher auch nicht direkt als Reparaturoperation verwendet werden. Die Eigenschaft, daß Constraints aus einer Menge zuvor abgesetzter Constraints zurückgenommen werden hat sie aber mit der allgemeinen Operation aus Def. 3.2.3 gemeinsam, so daß ich sie im folgenden auch als Reparaturoperation für asynchrone Constraintlöser verwenden werde, wobei dann immer sichergestellt wird, daß ihr Ergebnis nochmals auf Konsistenz geprüft wird.

Definition 5.1.6 (Suchalgorithmus) Ein Suchalgorithmus (ρ, l) ist gegeben durch eine ACS Reparaturoperation ρ und ein Labeling l, wobei für jede Constraintsequenz C gilt $\mathsf{vars}(l) = \mathsf{vars}(\varepsilon(C) \setminus \varepsilon(\rho(C)))$.

Durch Suchalgorithmen werden also immer über den Variablen neue Instantiierungen abgesetzt, deren alte Instantiierungen zurückgenommen wurden. An dieser Stelle werden dabei die ACS Reparaturoperation, also die Auswahl der zurückzunehmenden Instantiierungen, und die Wertauswahlfunktionen offengelassen. Suchalgorithmen werden durch Angabe dieser Operationen generisch definiert. Die Suchconstraints selbst können als generische Objekte, also als erweiterte ACS Constraints mit frei wählbaren Algorithmen, implementiert werden.

Definition 5.1.7 (Suchconstraint) Ein *Suchconstraint* (c, ρ) besteht aus einem ein ACS Constraint $c = (\overline{v}, T, (B, z, \rho'), E, \overline{h})$ und einer Reparaturoperation ρ, so daß gilt

1. Die Propagationsmethode p mit $(p, w) \in T$ mit der höchsten Priorität w ersetzt ρ' durch ρ und induziert die identische Propagationsfunktion.

2. Die Propagationsmethode p' mit $(p', w') \in T$ mit der niedrigsten Priorität w' ersetzt ρ durch ρ' und induziert die identische Propagationsfunktion.

3. Es existiert genau ein Labeling $l \in \mathsf{builtin}(c)$, so daß (ρ, l) ein Suchalgorithmus ist. Außerdem benutzt c keine weiteren Hilfsconstraints.

Bei der Ausführung eines Suchconstraints c wird also zunächst die vorherige Reparaturoperation durch eine ACS Reparaturoperation ersetzt, die an ein Labeling $l \in \mathsf{builtin}(c)$ so gekoppelt ist, daß ein Suchalgorithmus entsteht. Wenn bei der Ausführung von l eine Inkonsistenz entsteht, so nimmt ρ eine oder mehrere Instantiierungen aus $\mathsf{builtin}(l)$ zurück, woraufhin l neue Instantiierungen über den gleichen Variablen mit möglicherweise anderen Werten absetzt. Durch die nicht-deterministische Auswahl der zurückzunehmenden Constaints in ρ und der Wertauswahl in l werden so alternative Variablenbelegungen auf Konsistenz geprüft. Erst wenn keine neuen Instantiierungen mehr abgesetzt wurden, weil keine Inkonsistenzen mehr auftraten, wird die Reparaturoperation des Suchconstraints wieder durch die vorherige Operation ersetzt. Dann ist die aktuelle Variablenbelegung eine Lösung des im aktuellen Programm implementierten CSP. Wenn keine Lösung existiert oder aufgrund des Algorithmus nicht gefunden wird, so terminiert dieser Algorithmus nur dann, wenn das Labeling trotz aufgetretener Inkonsistenzen keine weiteren Instantiierungen mehr absetzt.

Fallstudie, Teil 13
Ein Suchconstraint für $\mathcal{M}$Plan setzt über jede seiner Variablen eine Instantiierung (vgl. Teil 12 der Fallstudie) ab. Während der Ausführung wird die Reparaturoperation des Solvers durch die des Suchconstraints ersetzt.

```
class Suchconstraint{
...
  // dispatcher der Propagationsmethoden
  void propagate(int x,Appointment[] apps){
    // zuerst: InconsistencyEventListener ersetzen
    if(x == 10){
      oldRepair = acs.getRepairOp();
      acs.setRepairOp(new SearchListener());
      return;}
    // dann: initiales Labeling absetzen
    if(x == 3){ labeling(apps);return;}
    // und zum Schluss: InconsistencyEventListener rueck-ersetzen
    if(x == 1){
       acs.setRerpairOp(oldRepair);
       return;}
  }
  // zum Sichern der Reparaturoperation des Solvers
  InconsistencyListener oldRepair = null;

  private void labeling(Appointment[] apps){
      // fuer jeden Termin eine AppInstantiation
      // direkt oder durch ein Hilfsconstraint absetzen:
      // (new AppInstantiation(vars[i])).post(acs);
      ...
```

```
  }
...
```

Zusätzlich stellt das Suchconstraint eine ACS Reparaturoperation zur Verfügung, die beim Auftreten von Fehlern Instantiierungen zurücknimmt und für die lokalen Variablen andere ausprobiert:

```
...
  class SearchListerner extends InconsistecyListener{
    ...
    // wird aufgerufen, nachdem Inkonsistenz e erzeugt
    // und ge-broadcast-et wurde
    void handleInconsistecy(InconsistecyEvent e){
      // Variablen, die mit der Inkonsistenz zu tun hatten
      Enumeration vars = e.getVariables();
      while(vars.hasMoreElements()){
        Variable aktu = vars.nextElement();
        // wenn aktu von this instantiiert wurde
        // und wahrscheinlich falsch ist
        if(myVars.contains(aktu) && errorLikely(aktu)){
          // Instantiierung von aktu zurücknehmen
          // neue Instantiierung erzeugen und absetzten
          ...
        }
      }
    }
  }
}
```

Diese Reparaturoperation kann die richtigen Entscheidungen, welche Instantiierungen zurückgenommen werden, treffen, was in der Funktion `errorLikely` *entschieden wird. Es kann aber nicht vorausgesetzt werden,daß dadurch alle Inkonsistenzen beseitigt werden, denn im* `InconsistencyEvent` *kann nicht festgestellt werden, welche Variableninstantiierungen verantwortlich für das Scheitern waren.*

5.2 Vollständigkeit und Terminierung

Zusammenfassung. Um den Suchraum eines CSP mit endlich vielen Variablen und endlichen Variablendomänen vollständig durchsuchen zu können sollte bei der Traversierung verhindert werden, daß gleiche Variablenbelegungen mehrfach erzeugt und untersucht werden. In diesem Abschnitt wird die Eigenschaft, daß die erzeugten Variablenbelegungen von Suchconstraints paarweise verschieden sind, deskriptiv definiert. Mit Suchalgorithmen, die auf diese Weise den gesamten Suchraum durchmustern, kann man jede existierende Lösung eines CSP finden, indem man ein entsprechendes Constraintprogramm in einem ACS ausführt. Existiert keine Lösung, so kann man dies ebenfalls mit einem solchen vollständigen Suchalgorithmus nachweisen.

Bei der nicht-deterministischen Suche nach Lösungen von CSPs sollte möglichst der gesamte, durch die Variablendomänen aufgespannte Suchraum, untersucht werden. Das Labeling eines Suchalgorithmus sollte dazu auf Inkonsistenzen, die aus vorherigen Instantiierungen resultieren, möglichst so reagieren, daß die gleiche Fehlentscheidung nicht noch einmal getroffen wird. Die Wertauswahlfunktionen der Instantiierungen eines Labeling in einem Suchconstraint sollten also möglichst bei jeder Ausführung solche Werte liefern, die in dieser Kombination mit den Werten anderer Variablen noch nicht untersucht wurden. Die Eigenschaft, daß ein Suchconstraint bei jedem Aufruf des Suchalgorithmus, also bei jeder Iteration von Reparatur und Labeling, eine neue Variablenbelegung induziert, wird zunächst rein deskriptiv definiert. Wie man ein solches Verhalten implementieren kann gebe ich in Abschnitt 5.3.2 an.

Definition 5.2.1 (explorativ) Ein Suchconstraint heißt *explorativ*, wenn alle Variablenbelegungen, die sein Labeling induziert, paarweise verschieden sind. Ein Suchalgorithmus heißt *explorativ*, wenn er bei jedem Aufruf die Instantiierung mindestens einer Variablen ändert.

Während der Suche werden vom Labeling eines Suchconstraints wiederholt neue Instantiierungen über gleichen Variablen abgesetzt. All diese Instantiierungen sind Hilfsconstraints des Suchconstraints, so daß potentiell unendlich viele Instantiierungen über jeder Variable abgesetzt werden können. Ist dies der Fall, so terminiert aber das Constraintprogramm, in dem das Suchconstraint abgesetzt wurde nicht (vgl. Lemma 3.4.3). Durch die Eigenschaft der Explorativität wird die Anzahl der möglichen Instantiierungen jeder Variablen auf die Anzahl der Permutationen von Variablenbelegungen mit Werten eingeschränkt. Bei endlichen Wertebereichen und endlichen Variablenmengen ergeben sich dadurch endlich viele verschiedene Instantiierungen für jede Variable. Ich gebe daher folgendes Lemma an, um dann die Terminierung von explorativen Suchconstraints daraus abzuleiten, daß es nur endlich viele Permutationen gibt.

Lemma 5.2.1 Jedes explorative Suchconstraint hat endlich viele Hilfsconstraints.

Beweis:
Laut Def. 5.1.3, Def. 5.1.2 und Def. 5.1.7.3 enthält die Menge $\mathsf{builtin}(s)$ für jedes Suchconstraint (s, ρ) außer genau einem Labeling ausschließlich Instantiierungen. Weil jedes Labeling laut Def. 5.1.3 nur Variablen mit endlichen Domänen enthält, und weil jedes Constraint laut Def. 2.1.3 nur endlich viele Variablen haben kann, gibt es auch nur endlich viele Permutationen von Variablenbelegungen. Weil die Variablenbelegungen eines Labeling, das von einem explorativen Suchconstraint induziert wird laut Def. 5.2.1 aber paarweise unterschiedlich sind, kann das Labeling auch nur endlich viele Variablenbelegungen induzieren. Da eine Variablenbelegung durch endlich viele Instantiierungen implementiert wird, gibt es auch nur endlich viele Instantiierungen, die ja die einzigen kritischen Hilfsconstraints des Suchconstraints sind.

□

Daraus ergibt sich, daß durch das Absetzen eines explorativen Suchconstraints die Anzahl der Hilfsconstraints eines Constraintprogramms genau dann endlich ist, wenn sie in dem Programm ohne das Suchconstraint endlich ist. Damit ist Bedingung 2. aus Lemma 3.4.3 für die Erweiterung terminierender Constraintprogramme mit explorativen Suchconstraints erfüllt, so daß die Terminierung des gesamten Programms gesichert ist, wenn die Methoden des Suchalgorithmus selbst terminieren.

Lemma 5.2.2 Wenn ein Constraintprogramm P terminiert, so terminiert seine Erweiterung um ein exploratives Suchconstraint (c, s) genau dann, wenn die Implementierungen von c, s und $\mathsf{builtin}(c)$ terminieren.

Beweis:
Folgt wegen Lemma 5.2.1 direkt aus Lemma 3.4.3 und Lemma 3.5.4, weil Labeling und Instantiierungen keine weiteren Hilfsconstraints verwenden und laut Voraussetzung selbst terminieren.

□

Wenn für ein gegebenes CSP eine Lösung gemäß Def. 2.1.12 existiert, so sollte diese auch durch die Ausführung eines Constraintprogramms, das das CSP implementiert, berechnet werden können. Mit einem explorativen Suchconstraint z.B. kann diese Lösung berechnet werden, denn wenn alle Propagationsmethoden des Suchconstraints terminiert haben, so sind all seine Variablen instantiiert und gleichzeitig alle Constraints erfüllt. Wird das Suchconstraint über alle Variablen des CSP abgesetzt, so kann das Ergebnis des Constraintprogramms als Lösung des CSP interpretiert werden, indem jeder Variable der einzige Wert ihrer Domäne zugeordnet wird.

Definition 5.2.2 (Lösung eines Constraintprogramms) Eine *Lösung* eines Constraintprogramms P ist ein Ergebnis $\mathsf{res}(P) = \langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n) \rangle$ für das gilt $\forall i \in 1..n :\mid \mathsf{dom}(v_i) \mid = 1$.

Bei der Verwendung explorativer Suchconstraints werden immer Lösungen für Constraintprogramme gefunden, wenn sie existieren (was mit Satz 3 gezeigt werden wird). Wenn aber keine Lösungen des zugrundeliegenden CSP existieren, wenn also ein inkonsistentes CSP implementiert wurde, so ist es bei der Verwendung von explorativen Suchconstraints möglich, die Nicht-Existenz von Lösungen nachzuweisen. Wenn alle möglichen Variablenbelegungen untersucht wurden und dabei immer Inkonsistenzen auftraten gibt es keine Lösung. In diesem Fall sollte das Suchconstraint terminieren und die Information weitergeben, daß es keine Lösung gibt. Dies könnte z.B. durch einen besonderen `InconsistecyEvent` beim Solver angezeigt werden. In diesem Fall sollte dann das Suchconstraint selbst zurückgenommen werden, was laut Abb. 3.9 auch alle Instantiierungen löscht, um einen wohldefinierten Zustand herzustellen, in dem das Suchconstraint nicht gilt. Es liegt dann nahe, das CSP des Constraintprogramms durch Rücknahme von Constraints so zu verändern, daß es lösbar wird.

Bei der Verwendung nicht-explorativer Suchconstraints kann eine solche Nicht-Existenz von Lösungen nicht nachgewiesen werden. Weil gleiche Variablenbelegungen mehrfach hergestellt und auf Konsistenz geprüft werden können. Man kann dann nie mit Sicherheit sagen, daß der gesamte Suchraum durchsucht wurde (vgl. Beweis zu Lemma 5.2.1). In solchen Suchconstraints empfiehlt es sich daher, eine andere Art der Terminierung herbeizuführen, wenn es wahrscheinlich ist, daß keine Lösung gefunden werden kann. In der ACS-Implementierung J.CP wird dies durch eine *timeout*-Funktion in nicht-explorativen Suchconstraints, die z.B. *local-search* Suchalgorithmen verwenden, bewerkstelligt. Wenn nach einer gewissen Zeit keine Lösung gefunden wurde, so terminiert der Algorithmus trotzdem. Auch in diesem Fall sollten dann alle Konsequenzen des Suchconstraints aus dem aktuellen Zustand entfernt werden, was sich am besten durch Rücknahme des Suchconstraint umsetzen läßt.

Solche Fälle, in denen Suchconstraints bzw. Suchalgorithmen aufgrund technischer Überlegungen terminieren, ohne eine Lösung gefunden zu haben, werden in inkrementellen Constraintverarbeitenden Systemen normalerweise explizit als Fehlerfall behandelt (z.B. in der CLP [FA97]).

Definition 5.2.3 (Scheitern) Wenn ein Suchconstraint terminiert und keine Lösung liefert, so *scheitert* es.

Nun kann gezeigt werden, daß ein Suchconstraint, ob explorativ oder nicht, immer dann Lösungen erzeugt, wenn es nicht explizit abgebrochen wurde, also scheiterte.

Satz 3 Wenn ein Suchconstraint *srch* in einem Constraintprogramm P terminiert und nicht scheitert, so ist sein Ergebnis eine Lösung von $\mathsf{constraints}(P) \setminus \{srch\}$.

Bemerkung 5.2.1
Mit dem Begriff "Lösung" ist im Satz eine Lösung des CSP (Def. 2.1.12) und nicht des Constraintprogramms (Def. 5.2.2) gemeint.

Beweis:
Wegen Def. 5.2.3 und Def. 5.2.2 erzeugt P laut Voraussetzung ein Ergebnis $\mathsf{res}(P) = \langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n) \rangle$, bei dem die Domäne jeder Variablen genau ein Element enthält. Dadurch ist der Vektor $\overline{d} \in \mathsf{dom}(v_1) \times ... \times \mathsf{dom}(v_n)$ als einziges Element des Kreuzprodukts eindeutig bestimmt. Es kann indirekt gezeigt werden, daß $\overline{d} \in \mathsf{sol}(P')$ (gemäß Def. 2.1.12) für $P' = seq(\mathsf{constraints}(P) \setminus \{srch\})$, wobei *seq* die Elemente einer Menge sequentialisiert, gilt:

Angenommen $\overline{d} \notin \mathsf{sol}(P')$

$$\overset{Def.\ 3.4.3}{\Rightarrow}$$

$$\exists c \in \mathsf{constraints}(P) : c \models \overline{d} \wedge \overline{d} \notin \mathsf{sol}(P')$$

$$\overset{Bem.\ 3.3.6, Def.\ 3.3.6.4}{\Rightarrow}$$

$$\mathsf{c} \text{ hat } \texttt{InconsistecyEvent} \text{ erzeugt}$$

$$\overset{Def.\ 5.1.5}{\Rightarrow}$$

$$\exists (z, A, \rho) \in \mathsf{solvers}(P), c' \in \mathsf{constraints}(P) : retract(c') \in \varepsilon(A)$$

$$\overset{Def.\ 3.4.3}{\Rightarrow}$$

$\langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n) \rangle$ ist kein Ergebnis, weil der Puffer A noch nicht leer ist.

□

Daraus ergibt sich, daß man auch nicht-triviale CSPs, die nicht durch die Herstellung einer Konsistenz (Def 2.2.2) lösbar sind, mit ACS lösen kann. Dazu muß das CSP als Constraintprogramm ausgeführt und ein Suchconstraint über seinen Variablen abgesetzt werden. In dieser Allgemeinheit, wenn also auch nicht-explorative Suchconstraints verwendet werden können, ist ACS aber nur ein unvollständiger Constraintlöser, denn ein schlechter nicht-explorativer Suchalgorithmus findet sicherlich nicht immer eine Lösung, selbst wenn welche existieren. Verwendet man einen explorativen Suchalgorithmus, so erhält man einen Constraintlöser, der immer genau dann ein Lösung findet, wenn eine existiert.

Korollar 5.2.1 Ein ACS, in dem ein terminierendes Constraintprogramm mit Suchconstraint ausgeführt wird, ist ein (unvollständiger) Constraintlöser.

Um einen vollständigen Constraintlöser zu erhalten, muß der Solver alle Lösungen beliebiger, implementierbarer CSPs finden können (Def. 2.2.3). Dies kann erreicht werden, indem man ein exploratives Suchconstraint erneut ausführt, nachdem es bereits eine Lösung gefunden hatte. Bei dieser erneuten Ausführung wird die Instantiierung einer Variable zurückgenommen und eine alternative Instantiierung abgesetzt. Entsteht dadurch eine Inkonsistenz, so werden nur Variablenbelegungen erzeugt, die vorher noch nicht untersucht wurden. Dadurch entstehen keine Zyklen, in denen gleiche Lösungen mehrfach erzeugt werden. In ACS wird das durch einen explorativen Suchalgorithmus implementiert, der aufgerufen wird, nachdem eine Lösung gefunden wurde. Der Algorithmus erzeugt ein alternatives Labeling und verhindert die Erzeugung von bereits untersuchten Variablenbelegungen.

Definition 5.2.4 (Iterierbares Suchconstraint) Ein *iterierbares Suchconstraint* (s, n) ist gegeben durch ein Suchconstraint $s = (c, \rho)$ und einen explorativen Suchalgorithmus $n = (l, \rho)$. Wobei die induzierten Variablenbelegungen von l und c paarweise verschieden sind.

Um mehrere Lösungen für ein Constraintprogramm zu finden, muß man beim ACS ein iterierbares Suchconstraint absetzen, das dann als Objekt referenziert werden kann. Das Anstoßen des Suchalgorithmus kann dann mit einer Methode dieses Objekts implementiert werden. Dies wird in anderen Constraintlösern ähnlich implementiert, so kann man in CLP-Systemen wie SICStus [SICS] zu jedem *labelling* alternative Lösungen erzeugen, indem eine alternative Ableitung gesucht wird. In Constraintlösern für imperative Sprachen oder in imperativen Schnittstellen von CLP-Systemen wird meist auch eine explizite Operation `nextSolution` zur Verfügung gestellt, die genau diese Funktionalität umsetzt. Die auf diese Weise erzeugten unterschiedlichen Lösungen können im Anwendungsprogramm verarbeitet werden.

Definition 5.2.5 (Erzeugte Lösungen) Die Menge der mit einem iterierbaren Suchconstraint (s, n) erzeugten Lösungen in einem Constraintprogramm P ist gegeben durch die Variablenbelegungen der Zustände, die nach der Ausführung von n Ergebnisse von P waren.

Mit explorativen Suchconstraints kann man, wie bereits beschrieben, alle Lösungen eines CSP erreichen, weil potentiell unendlich viele Variablenbelegungen untersucht werden können. Ist das Suchconstraint zusätzlich iterierbar, so kann man solange weitere Lösungen erzeugen, bis der Suchalgorithmus scheitert. Solche iterierbaren und explorativen Algorithmen werden in der Literatur oft als vollständige Suchalgorithmen bezeichnet [MS98].

Satz 4 Mit einem explorativen und iterierbaren Suchconstraint kann man alle Lösungen eines terminierenden Constraintprogramms erzeugen.

Beweis:
Laut Satz 2 und 3 sind alle erzeugten Lösungen in $\mathsf{sol}(\mathsf{constraints}(P))$ enthalten. Zu zeigen bleibt, daß alle Elemente von $\mathsf{sol}(\mathsf{constraints}(P))$ als Lösungen erzeugt wurden. Da der Suchalgorithmus eines iterierbaren Suchconstraints laut Def. 5.2.4 explorativ ist, erzeugt er potentiell unendlich viele paarweise verschiedene Lösungen (Def. 5.2.5). Weil die Domänen aller $v \in \mathsf{vars}(P)$ laut Def. 5.1.3 endlich sind, gibt es aber nur endlich viele induzierte Variablenbelegungen der durch das Suchconstraint abgesetzten Labelings, so daß auch alle Lösungen erzeugt werden.

□

Daraus ergibt sich laut Def. 2.2.3 das folgende Korollar.

Korollar 5.2.2 Wird in einem Asynchronen Constraintlöser ACS ein vollständiges Suchconstraint abgesetzt, so ist ACS ein vollständiger Constraintlöser.

5.3 Implementierung für ACS

Zusammenfassung. In diesem Anschnitt werden Techniken angegeben, die verwendet werden können, um bekannte Suchalgorithmen in ACS Suchconstraints zu implementieren. Es wird erläutert, wie man Propagationsmethoden zur Verwendung in chronologischen Algorithmen sequentialisieren kann. Weiterhin wird eine Technik angegeben, um bereits bekannte Variablenbelegungen zu verwalten, was zur Definition vollständiger bzw. explorativer Algorithmen notwendig ist. Schließlich wird beschrieben, an welchen Stellen und auf welche Weise man bei ACS Anwendungen die bei der Suche erforderlichen nichtdeterministischen Entscheidungen treffen kann.

Aus den vorherigen Abschnitten ergibt sich, daß Suche in ACS als spezielles Suchconstraint integriert wird. Diese Constraints kapseln Algorithmen, die Instantiierungen absetzen und zurücknehmen, bis eine Lösung für das implementierte CSP gefunden werden konnte oder die Suche scheiterte. Instantiierungen sind dabei selbst Constraints, deren Auswirkung dementsprechend bei den anderen Constraints propagiert wird. Bei der Suche wird die Konsistenz von nichtdeterministisch gewählten Werten der Instantiierungen also durch *forward-checking* gesichert. In diesem Abschnitt werden darauf aufbauend konkrete Techniken angegeben, wie man aus der Literatur bekannte Suchalgorithmen in ACS Suchconstraints implementieren kann.

5.3.1 Sequentielle Algorithmen

In der Definition von Suchconstraints (Def. 5.1.7) wird durch die Verwendung unterschiedlicher Prioritäten für die unterschiedlichen Propagatoren implizit eine bestimmte Reihenfolge für die Ausführung der Aufgaben des Constraints definiert. Bei Suchconstraints muß zuerst die Reparaturoperation des Solvers ersetzt werden, danach wird der Suchalgorithmus ausgeführt und zuletzt wird die ursprüngliche Reparaturoperation wieder dem Solver zugeordnet. Diese Reihenfolge ist dringend zur Umsetzung von Suchconstraints erforderlich. In vielen Suchalgorithmen spielt ebenfalls die Reihenfolge, in der Entscheidungen getroffen beziehungsweise zurückgenommen werden, eine wichtige Rolle. Beim klassischen chronologischen *backtracking* [CLR89] zum Beispiel ist die Einhaltung einer globalen Reihenfolge der Ausführung von Operationen und Umkehroperationen notwendig. Algorithmen, die eine solche globale Reihenfolge erfordern, heißen sequentielle oder chronologische Algorithmen [HH89].

Durch die Asynchronität und die Möglichkeit, mehrere Solver nebenläufig zu verwenden, existiert bei der Ausführung von Constraintprogrammen (Def. 3.3.4 bzw. Abb 3.3) beim ACS keine solche globale Reihenfolge. Man kann bei der Verwendung mehrerer Solver in einem Constraintprogramm nicht angeben, in welcher Reihenfolge Propagationsmethoden ausgeführt werden sollen. Auch im Nachhinein ist es nicht möglich festzustellen, in welcher Ordnung bestimmte Methoden ausgeführt wurden. Die Ausführung der Propagationsmethoden geschieht also bei der Verwendung mehrerer Solver echt nebenläufig. Wird allerdings nur ein Solver verwendet, so existiert eine eindeutige Reihenfolge, in der die Methoden ausgeführt wurden.

Daraus ergibt sich als Voraussetzung für die Implementierung sequentieller Suchalgorithmen in ACS, daß sie vollständig innerhalb eines Solvers ausgeführt oder explizit synchronisiert werden, das heißt, das Labeling bzw. die Instantiierungen, die Reparaturoperation und die Konsistenzüberprüfung müssen in einem Solver ausgeführt werden, um überhaupt eine Reihenfolge festlegen zu können. Das Labeling und die Instantiierungen sind per Definition nur in einem Solver ausführbar, weil diese Hilfsconstraints des Suchconstraints sind und somit laut Def. 3.3.6.2 in dem selben Solver ausgeführt werden wie das Suchconstraint. Die Konsistenzprüfung wird allerdings in den Propagationsmethoden verschiedener Constraints ausgeführt, die potentiell unterschiedlichen Solvern zugeordnet sein könnten. Werden beim *forward-checking* einer Instantiierung eines Suchconstraints also Constraints in anderen Solvern ausgeführt, so muß sichergestellt werden, daß ihre Propagationsmethoden ausgeführt wurden bevor die Instantiierung als konsistent gilt. Dies kann durch den Zuverlässigkeitswert des Solvers implementiert werden, die Instantiierung gilt in einem Solver erst dann als konsistent, wenn der Solver anderer relevanter Constraints einen bestimmten Zuverlässigkeitswert erreicht hat. Verwendet man in einem Constraintprogramm mit einem sequentiellem Algorithmus nur genau einen Solver, so ist die Existenz einer Reihenfolge von vorneherein gesichert.

Schließlich bleibt noch zu klären, wie in einem Solver eine Reihenfolge für die Methodenaufrufe während eines sequentiellen Algorithmus umgesetzt werden kann. Wie am Anfang dieses Abschnitts bereits angedeutet wurde, werden hierzu die unterschiedlichen Prioritäten der Propagationsmethoden verwendet. Wenn ein Constraint mit mehreren Propagationsmethoden abgesetzt wird, so sollten die Aufgabenpuffer der Solver immer so implementiert sein, daß jeweils die Aufgabe mit der höchsten Priorität zuerst ausgeführt wird. Um die Ausführung von Propagationsmethoden unterschiedlicher Constraints zu sequentialisieren, muß man außer den Prioritäten noch eine explizite Synchronisation implementieren. Dies geschieht am besten, indem ein Constraint A, das nach einem Constraint B ausgeführt werden soll, in der letzten Propagationsmethode von B abgesetzt wird. Eine solche "letzte“ Propagationsmethode kann in jedem Constraint implementiert werden, indem ihr die niedrigste Priorität zugeordnet wird.

In Abschnitt 5.4.1 werde ich am Beispiel des chronologischen *backtracking*-Algorithmus angeben, wie das Absetzen und Zurücknehmen von Instantiierungen in ACS zeitlich synchronisiert und sequentialisiert werden kann.

Fallstudie, Teil 14

Die `labeling`*-Methode aus Teil 13 kann für jeden Termin einen Agenten erzeugen, der das entsprechende* `AppInstantiation`*-Constraint absetzt:*

```
class LabelingAgent extends Constraint{
  propagate(int x, Appointment app, LabelingAgent prev,next){
    if(x = 2){
       // in anderen Solvern, die durch app betroffen sein koennen
       Enumeration otherSolvers = app.getSolvers();
       while(otherSolvers.hasMoreElements())
          // jeweils vollstaendige Propagation abwarten
          otherSolvers.nextElement().awaitReliability(MAX);
       instantiate(app);
    }
  }

  private void instantiate(app){
    (new AppInstantiation(app).post(acs);
    ...
    // Den Agenten für next oder prev aktivieren:
    // next.post(acs); oder prev.backtrack();

  }
}
```

Der Agent mit Priorität 2 wird immer nach den Propagationsmethoden aller anderen Constraints und vor der Beendigung des Labeling (vgl. Teil 13) im eigenen Solver ausgeführt. Durch den Methodenaufruf `awaitReliability(MAX)` *wird der Prozeß dieses Solvers solange angehalten, bis jeweils die anderen Solver die Propagation abgeschlossen haben. Hierdurch wird also die Propagation im verteilten System explizit synchronisiert. Durch die Abhängigkeiten von Constraints (Def. 2.1.2 kann aus* `app` *eine Aufzählung aller Solver berechnet werden, die direkt oder indirekt durch die Instantiierung von* `app` *betroffen sind (Def. 3.5.1).*

5.3.2 Explorative Suchalgorithmen

Explorative Suchalgorithmen erzeugen in jeder Situation nur Variablenbelegungen, die vorher nicht erzeugt wurden (Def. 5.2.1). Der Vergleich von Variablenbelegungen dafür kann aber nicht lokal z.B. in den Variablen, indem deren betrachtete Werte abgespeichert werden durchgeführt werden, weil zur Herstellung aller Permutationen auch gleiche Werte in einzelnen Variablen mehrfach betrachtet werden müssen. Die Möglichkeit, bereits betrachtete Variablenbelegungen zentral abzuspeichern um zu verhindern, daß sie erneut erzeugt werden, wird aus Gründen der Performanz und des Speicherbedarfs nicht angewandt. Außerdem sollte mit Blick auf die verteilte Anwendung von Suchalgorithmen auf solche komplexen zentralen Strukturen verzichtet werden, um so wenig wie möglich Kommunikation erforderlich zu machen.

Explorative Suchalgorithmen für nebenläufige Anwendungen [YD98, Dur99] verwenden wenig globale Information um die betrachteten Werte lokal in den Variablen verwalten zu können. In allen vollständigen Suchalgorithmen für verteilte Probleme wird eine globale partielle Ordnung über den Variablen festgelegt. Aus dieser Ordnung kann ein Algorithmus zur Aufzählung aller Permutationen abgeleitet werden. Ein solcher Algorithmus wird auch beim chronologischen *backtracking* verwendet, um dessen Vollständigkeit bei endlichen Variablendomänen zu garantieren. Beim ACS kann eine solche Ordnung im Suchconstraint selbst definiert werden, denn von ihm aus kann die Instantiierung aller Variablen zentral gesteuert werden. Dadurch sind sowohl sequentielle als auch nebenläufige vollständige Suchalgorithmen implementierbar.

Angenommen, ein Suchconstraint definiert eine Ordnung auf ihren Variablen, dann kann für jede Variable ein Agent erzeugt werden, der den Agent seines Vorgängers und den seines Nachfolgers kennt. Diese Referenzen können sich evtl. während der Suche auch dynamisch ändern, wenn die Explorativität dabei erhalten bleibt (wie z.B. beim *weak commitment search* [YD98, YH00]). Wenn dann jeder Agent die betrachteten oder die noch nicht betrachteten Werte der Domäne seiner Variablen verwaltet, so kann er jederzeit Instantiierungen mit neuen Werten über seiner Variablen absetzen. Dies geschieht immer dann, wenn der Vorgänger seine Variable so instantiiert hat, daß keine Inkonsistenz entstand. Tritt eine Inkonsistenz beim *forward-checking* einer Instantiierung auf, so muß die letzte Instantiierung zurückgenommen werden und eine neue Instantiierung mit einem alternativen Wert abgesetzt werden. Sind keine alternativen Werte mehr bekannt, so wird *backtracking* eingeleitet. Dazu werden alle Werte des Agenten, der keine Alternativen mehr hat, als "unbetrachtet" markiert und der vorherige Agent benachrichtigt, seine Variable anders zu instantiieren. Wenn dabei keine Inkonsistenz auftritt, werden die als "unbetrachtet" markierten Werte, die in einem anderen Zusammenhang schon betrachtet wurden, erneut durchgemustert.

In Abschnitt 5.4.1 wird ein sequentieller Algorithmus angegeben, der mit diesem Vorgehen das bekannte chronologische *backtracking* [CLR89] simuliert. Dieser Suchalgorithmus für monolithische Problemlöser kann also durch die Ordnung von Variablen und durch die dadurch festgelegte Kommunikation von Agenten auch beim asynchronen Constraintlösen angewandt werden.

Fallstudie, Teil 15

Durch die Synchronisation der Instantiierungen (Teil 14 der Fallstudie) kann eine eindeutige Reihenfolge der Instantiierung der Variablen angegeben werden. Dazu wird im Labeling eine Reihenfolge für die Instantiierungen der Termine festgelegt, aus der sich aus der Implementierung der `AppInstantiation` *eine Reihenfolge der Variableninstantiierungen ableiten läßt. Danach kann der Agent für die erste Variable gestartet werden.*

```
...
  private void labeling(Appointment[] apps){
    // ein Vector fuer alle Constraintvariablen
    Vector vars = new Vector(apps * 4);
    // ein Array fuer alle LabelAgenten der Termine
    LabelAgent[] lab = new LabelAgent[apps.legth];
```

```
    for(int i = 0; i < apps.length; i++){
      // Variablenreihenfolge speichern
      vars.add(apps[i].getTag());
      vars.add(apps[i].getZeit());
      vars.add(apps[i].getDauer());
      vars.add(apps[i].getOrt());
      // Labeling-Agenten fuer alle Termine erzeugen
      lab[i] = new LabelAgent(app);
    }
    ...
  }
```

In einem Variablen-indizierten Vektor von Domänen können dann immer alle bereits benutzten Werte abgespeichert werden:

```
  Domain[] usedValues = new Domain[vars.length];
```

Diese werden dann von den `LabelAgent`*-Objekten immer aktualisiert, wenn eine Instantiierung abgesetzt wurde. So wird der entsprechende Wert hier eingetragen, beim backtracking werden alle Werte als "unnbenutzt" gekennzeichnet, also* `usedValue` *der leeren Domäne zugewiesen.*

5.3.3 Nichtdeterministische Entscheidungen

Wie in Abschnitt 5.1 bereits erwähnt wurde, sind bei der Ausführung von Suchalgorithmen beim ACS mehrere nicht-deterministische Entscheidungen zu fällen. Die Reihenfolge, in der die verschiedenen Werte einer Variablen untersucht werden (Wertauswahl) steht in Suchalgorithmen immer an [HE80]. Bei explorativen Algorithmen ist, wie im vorherigen Abschnitt beschrieben, außerdem eine Reihenfolge zu wählen, in der die Variablen instantiiert werden (Variablenauswahl). Bei nicht-explorativen Algorithmen ist festzulegen, welche Instantiierungen als Reaktion auf eine Inkonsistenz zurückgenommen werden sollen (Constraintrücknahme).

Wertauswahl

Die nichtdeterministische Wertauswahlfunktion *indomain* aus Def. 5.1.1 wird in den Propagationsmethoden von Instantiierungen (Def. 5.1.2) aufgerufen. Sie kann beim ACS daher entweder in den Variablen selbst oder in der entsprechenden Propagationsmethode implementiert werden. An beiden Stellen steht die Information zur Verfügung, welche Werte in der aktuellen Domäne der Variable enthalten sind. Bei explorativen Algorithmen ist dabei darauf zu achten, daß man bei mehrmaliger Ausführung der Wertauswahlfunktion immer andere Werte erhält bzw. Werte als "betrachtet" oder "unbetrachtet" markieren kann (vgl. Abschnitt 5.3.2). Unterschiedliche Wertauswahlfunktionen können zu erheblichen Performanzunterschieden beim Lösen von CSPs führen, was in Abschnitt 7.2 genauer untersucht werden wird. In den meisten Fällen ist es besonders günstig, zufällig gewählte Werte aus der Domäne als Resultat der Wertauswahl zu verwenden.

Variablenauswahl

Bei explorativen Suchalgorithmen muß die Reihenfolge, in der die Variablen instantiiert werden a priori festgelegt werden. Algorithmen, die eine dynamische Umordnung der Variablen benutzen, sind zur Zeit für ACS nicht implementiert. Bei nicht-explorativen Algorithmen ist keine feste Reihenfolge zur Lösungsfindung notwendig, so daß bei solchen Algorithmen beliebige Strategien angewandt werden können. Die Reihenfolge der Variableninstantiierungen kann willkürlich im Constraintprogramm oder im Suchconstraint definiert werden. Dabei empfiehlt es sich in vielen Fällen, die Variablen mit den wenigsten möglichen Werten zuerst zu instantiieren (die sog. *first-fail*-Strategie [HE80]). In der Literatur wurden aber auch andere Variablenauswahlstrategien untersucht und für spezielle Anwendungen verglichen. Wie man eine festgelegte Reihenfolge in einer konkreten Anwendung einhalten kann, wurde in Abschnitt 5.3.1 bereits beschrieben. Man kann z.B. Agenten für verschiedene Variablen synchronisieren.

Constraintrücknahme

Die Frage, welche Instantiierung bei der Suche als Reaktion auf eine Inkonsistenz zurückgenommen werden soll (Def. 5.1.5), ist bei explorativen Algorithmen durch die Ordnung der Variablen gegeben. Bei nicht-explorativen Algorithmen muß hierzu eine Heuristik verwendet werden, die aus den `InconsistencyEvent`-Objekten, die beim Entdecken von Inkonsistenzen erzeugt werden, eine Entscheidung ableitet. Man kann zum Beispiel immer die Instantiierungen bevorzugt zurücknehmen, deren Variablen in den meisten unerfüllbaren Constraints vorkamen. Dies ist aber nicht unbedingt das beste Vorgehen, weil durch die Propagation oft Fehlentscheidungen bei der Instantiierung einer Variablen erst in entfernten Constraints auftauchen, so daß nicht mehr nachzuvollziehen ist, wo eigentlich die Fehlentscheidung getroffen wurde. In *local-search*-Algorithmen werden daher Bewertungsfunktionen betrachtet, die die möglichen Änderungen an einem inkonsistenten Zustand vergleichen. Dazu muß man aber parallel mehrere unterschiedliche Reaktionen auf eine Inkonsistenz ausführen und deren Resultat vergleichen. Schließlich können auch zufällig Instantiierungen zu Rücknahme ausgewählt werden, was zur Vermeidung von Zyklen, die ja bei nicht-explorativen Algorithmen nicht erkannt werden können, ein geeignetes Mittel ist.

5.3.4 Generische Suchconstraints

Die Suchconstraints in der aktuellen Version von J.CP sind so implementiert, daß sie unabhängig vom Constraintsystem ihrer Variablen verwendet werden können. Die im nächsten Abschnitt vorgestellten Constraints sind demnach in dem Constraintsystemunabhängigen Java-Paket `jcp` enthalten. Voraussetzung für diese generische Implementierung war die Möglichkeit, mit Variablen Instantiierungsconstraints zu erzeugen, die die Variable in Beziehung zu einem Wert ihrer Domäne setzten. Dazu ist in der abstrakten Prototypklasse für Variablen eine Methode vorgesehen, die ein Objekt einer Implementierung des allgemeinen *interfaces* für Instantiierungen erzeugt. Diese Methode muß von Variablen jedes Constraintsystems implementiert werden, weil diese jeweils Spezialisierungen der abstrakten Klasse für Variablen sind.

Die so erzeugten Instantiierungs-Objekte, die die Spezifika ihres Constraintsystems nutzen können, werden dann vom Suchalgorithmus abgesetzt bzw. zurückgenommen, um eine Lösung zu finden. Dadurch kann man die Suchconstraints auch als heterogene Constraints über Variablen verschiedener Constraintsysteme verwenden. In der $\mathcal{M}$Plan Anwendung wird z.B. beim Festlegen eines Termins ein Labeling für alle Variablen, die den Termin beschreiben abgesetzt. Damit wird ein Suchconstraint über *fd*- und *enum*-Variablen abgesetzt, das durch Iteration über die Elemente der Domänen eine Lösung findet, indem lediglich unterschiedliche ungetypte Werte (Instanzen der Klasse `Object`) auf Konsistenz untersucht werden.

Die Implementierung von explorativen Algorithmen bzw. die der Wertauswahlfunktion, mit der verschiedene Werte der Variablendomänen successive untersucht werden können, muß bei dieser generischen Implementierung besonders beachtet werden. Die Variablen müssen eine Möglichkeit zur Verfügung stellen, alle Werte ihrer Domäne durch mehrmaligen Aufruf einer *indomain*-Methode zu liefern. In J.CP ist diese Funktionalität so implementiert, daß jede Variable eine Instanz der Java-Klasse *Iterator* erzeugen kann, mit der eine Menge repräsentiert werden kann. Dieser Iterator stellt Methoden zur Verfügung, mit dem man über alle Elemente der Variablendomäne iterieren kann. Die Klasse dieser Elemente ist die allgemeinste Java-Klasse `Object`, so daß effektiv keine Typisierung verwendet wird.

5.4 J.CP Suchalgorithmen

Zusammenfassung. In diesem Abschnitt wird die Implementierung der Suchconstraints, die in der aktuellen Implementierung J.CP zur Verfügung stehen, dargestellt. Es sind ACS Constraints, die um einen *EventListener* zur Verarbeitung von Inkonsistenzen erweitert sind. Die Instantiierungen der Variablen wird durch Agenten gesteuert, die jeweils die Werte zur Instantiierung wählen und das Absetzen und Zurücknehmen der Instantiierungen koordinieren. Die vorgestellten Suchconstraints benutzen unterschiedliche Algorithmen (chronologisches *backtracking* und *local-search*), deren besondere Anwendungsgebiete jeweils diskutiert werden.

Suchconstraints beim ACS werden wie andere Constraints auch, in einem Solver mit der Methode `post` abgesetzt. Bei ihrer Ausführung wird die Reparaturoperation (Event-Listener) des Solvers durch die jeweilige ACS Reparaturoperation ersetzt. Iterierbare Suchconstraints stellen zusätzlich eine Methode zur Generierung weiterer Lösungen zur Verfügung (`nextSolution`). Die Architektur einer constraintbasierten Anwendung ist in Abb. 5.2 als Erweiterung der Systemarchitektur aus Abb. 4.1 dargestellt.

5.4.1 Chronologisches *backtracking*

Beim chronologischen *backtracking* wird vor der Ausführung der Suche eine totale Ordnung über allen zu instantiierenden Variablen und den Werten ihrer Domänen festgelegt. Diese beiden Ordnungen werden verwendet, um die alternativen Instantiierungen für jede einzelne Variable vollständig zu untersuchen.

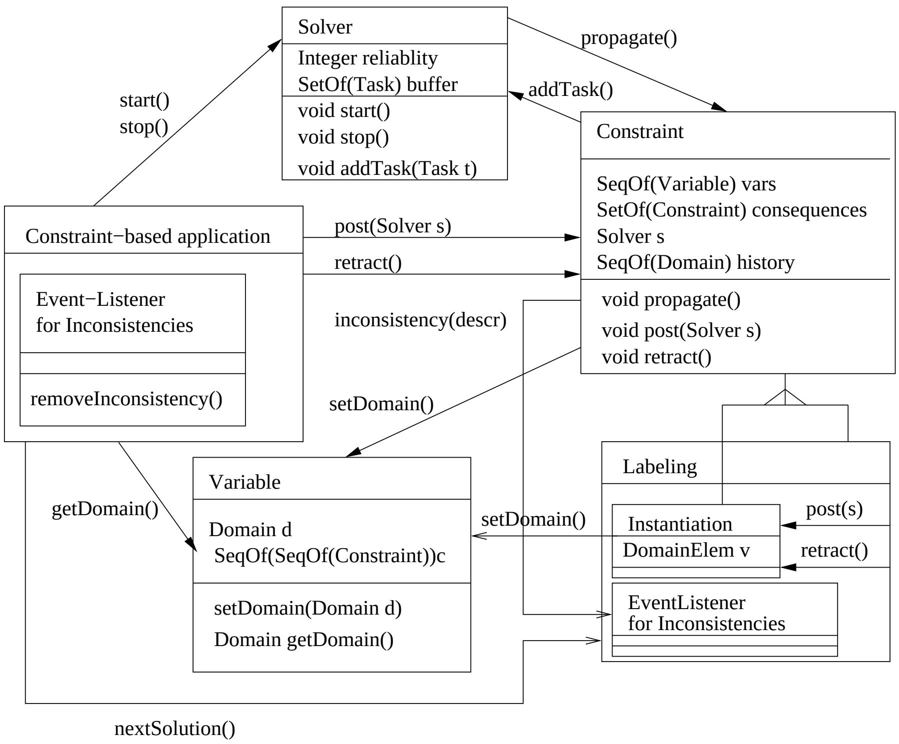

Abbildung 5.2: Architektur einer ACS Anwendung mit einem Suchconstraint “Labeling“.

Es werden der Reihe nach für jede Variable jeder ihrer Werte zur Instantiierung verwendet, bis man eine bzw. die nächste Lösung gefunden hat oder den gesamten Suchraum durchsucht hat. Der Algorithmus ist vollständig, denn durch die totale Ordnung ist bei jeder aktuellen Instantiierung bekannt, welche Instantiierungen vorher untersucht wurden. Dadurch kann verhindert werden, daß gleiche Variablenbelegungen mehrfach untersucht werden.

Chronologisches *backtracking* ist ein Standardalgorithmus zum Problemlösen, der in der Literatur (z.B. [CLR89]) nachzulesen ist. Der Algorithmus erfordert eine eindeutige Reihenfolge, in der die Instantiierungen ausgeführt werden. Außerdem muß nach jeder Instantiierung festgestellt werden, ob durch sie keine Constraints verletzt wurden, bevor die nächste Variable instantiiert werden kann. Inkonsistenzen werden also beim *labeling* immer nur durch die letzte instantiierte Variable verursacht. Die Constraintrücknahme dieser Constraints, die durch die ACS Reparaturoperation ausgeführt wird, ist besonders schnell durchzuführen, denn es existieren keine verwendenden Constraints (vgl. Def. 3.5.2 und Abb. 3.9), die betrachtet werden müßten.

Allerdings müssen das Suchconstraint und seine Hilfsconstraints, die chronologisches *backtracking* für ACS implementieren, mit den in Abschnitt 5.3.1 vorgestellten Techniken explizit sequentialisiert werden. Dazu wird für jede Variable ein Agent gestartet, der die Instantiierungen absetzt und zurücknimmt. Diese Agenten sind als ACS Constraints implementiert, deren Priorität niedriger ist als die aller "normalen" ACS Constraints. Dadurch werden sie nur aktiv, wenn keine Propagationsmethoden mehr zur Konsistenzprüfung auszuführen sind und wenn ihre Ausführung nicht bei der Verarbeitung einer Inkonsistenz verhindert wird. In Abb. 5.3 wird ein Ausschnitt des Klassendiagramms für diesen Algorithmus dargestellt.

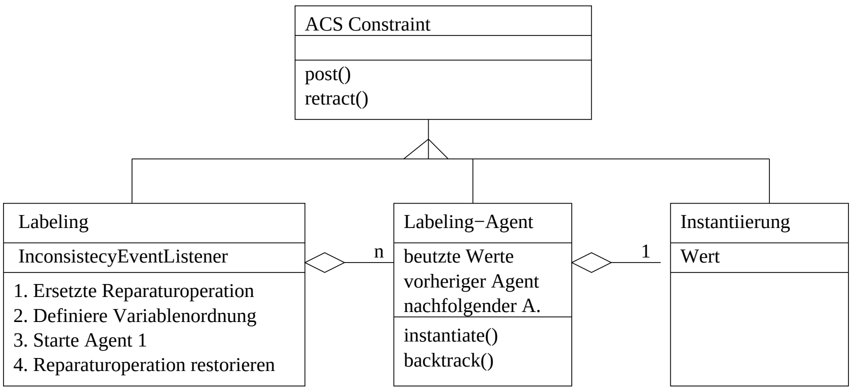

Abbildung 5.3: Klassendiagramm der ACS Implementierung von chronologischem *backtracking*. Die Methoden des Labeling werden so priorisiert, daß sie in der angegeben Reihenfolge ausgeführt werden. Bevor die Reparaturoperation rückersetzt wird, müssen alle Agenten ihre Variable erfolgreich instantiiert haben.

Die Labeling-Agenten jeder Variable verwalten die Werte, mit denen die Variablen jeweils schon instantiiert wurden, so daß die Explorativität des Algorithmus gesichert ist. Die Reihenfolge, in der die Werte verwendet werden, wird dabei durch eine entsprechende Methode der Variablen-Objekte festgelegt, denn in den generischen Suchalgorithmen kann keine interne Struktur der Domänen verwendet werden. Verschiedene Arten von Variablendomänen stellen verschiedene Iteratoren für ihre Elemente zur Verfügung, die von den generischen Suchconstraints bzw. deren Agenten aufgerufen werden. In Variablen mit endlichen Domänen ganzer Zahlen z.B. stehen in J.CP drei verschiedene Iteratoren zur Auswahl: aufsteigend, absteigend oder zufällig werden die verschiedenen Werte der Domäne durch eine Wertauswahlfunktion geliefert.

Die Ausführung dieses Suchconstraints mit seinen Hilfsconstraints (Labeling-Agenten und Instantiierungen) wird in Abb. 5.4 an einem Beispiel dargestellt. Das Labeling startet einen Agenten für die erste Variable. Dieser wählt einen Wert für seine Variable und setzt eine entsprechende Instantiierung ab. Diese Instantiierung wird dann durch die erneute Ausführung abhängiger Constraints in anderen Domänen propagiert. Danach ist der Agent für die nächste Variable an der Reihe und instantiiert seine Variable.

Ergibt sich dabei eine Inkonsistenz, so muß *backtracking* eingeleitet werden. Wie es in Abb. 5.5 dargestellt ist, wird dazu die letzte Instantiierung zurückgenommen. Außerdem werden die Werte dieser letzten Variablen als "unbenutzt"

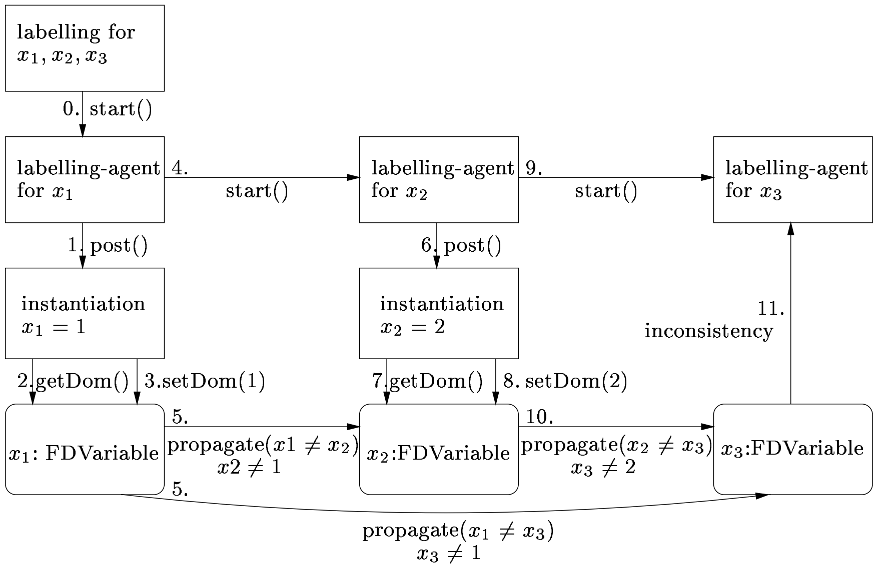

Abbildung 5.4: Chronologisches *backtracking* in ACS für das CSP $(\langle\{1,2\},\{2,3\},\{1,2\}\rangle,\langle x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3\rangle)$. Die Variable x_1 wird mit 1 instantiiert. Wenn keine Propagation von x_3 nach x_2 durchgeführt wird, so muß auch für x_2 nichtdeterministisch ein Wert gewählt werden. Bei der Instantiierung von x_2 mit 2 entsteht eine Inkonsistenz, die die Variable x_3 ihrem Agenten mitteilt. Normalerweise kann durch *forward-checking* bereits nach der Instantiierung einer Variable in diesem CSP eine Lösung berechnet werden, zu Demonstrationszwecken sei dies hier aber dadurch ausgeschlossen, daß von x_2 zu x_3 keine Propagation stattfindet.

markiert, so daß sie bei einer alternativen Instantiierung der vorherigen Variable erneut verwendet werden können. Wenn die Instantiierung mit einem Wert durch Propagation keine Inkonsistenzen erzeugt, so kann die nächste Variable instantiiert werden. Wenn es keine solche nächste Variable gibt, so sind alle Variablen instantiiert und es traten keine Inkonsistenzen bei der letzten Variablenbelegung auf. Dies bedeutet aber, daß die aktuellen Variablendomänen jeweils genau einen Wert enthalten und somit die Domänen der Variablen zusammengenommen eine Lösung (Def. 5.2.2) des Constraintprogramms definieren.

Wegen der Eigenschaft der Vollständigkeit ist chronologisches *backtracking* vor allem dann als Suchconstraint zu benutzen, wenn alle Lösungen eines Problems deterministisch gefunden werden sollen. Durch die Synchronisierung der Propagationsmethoden ist die Ausführung dieses Algorithmus (bei Verwendung einer deterministischen Wertauswahlfunktion) deterministisch, so daß man bei jeder Ausführung gleiche Werte nach den gleichen Berechnungen erhält. Allerdings ist der Algorithmus nicht gut für verteilte Anwendungen geeignet, weil die Synchronisation der Propagation mit den Instantiierungen viel Kommunikation erfordert. Wenn Constraints in unterschiedlichen Prozessen erzeugt wurden, so wird das Suchconstraint hierdurch sehr ineffizient. Das klassische *backtracking* paßt wegen seiner Sequentialität nicht unbedingt zum asynchronen Ausführungsmodell von ACS. Das Erzwingen von Reihenfolgen macht Be-

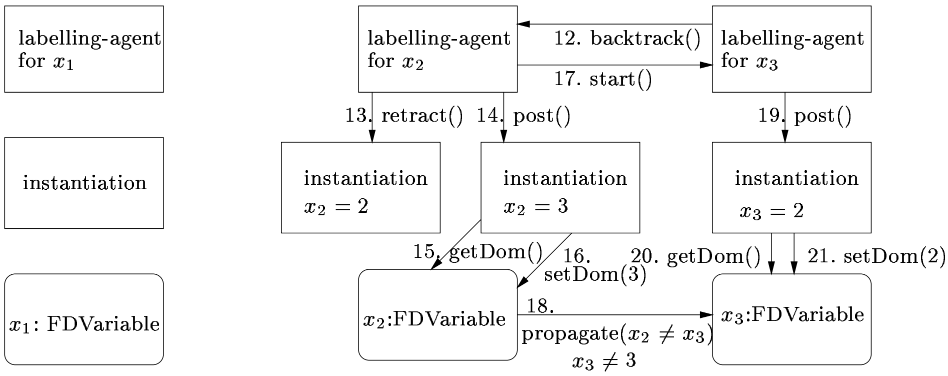

Abbildung 5.5: Beseitigung der Inkonsistenz aus Abb. 5.4 durch *backtracking*.

rechnungen erforderlich, die in sequentiellen Programmen nicht notwendig sind. Allerdings kann man mit einer solchen Implementierung klassische CSPs wie *benchmark*-Programme zum Beispiel ausführen, um J.CP mit anderen Solvern zu vergleichen.

Fallstudie, Teil 16

Durch die Sequentialisierung der Instantiierungen (Teil 14) kann mit der Verwaltung bereits untersuchter Werte (Teil 15) ein vollständiger Suchalgorithmus für $\mathcal{M}$*Plan analog zu Abb. 5.4 und Abb. 5.5 implementiert werden. Wenn der* `LabelingAgent` *der ersten Variable keine weiteren Werte mehr zur Auswahl hat, so ist das Labeling gescheitert, d.h. es gibt keine Lösung für das definierte CSP:*

```
class LabelAgent extends Constraint{
...
  // der Termin, fuer den dieser Agent zustanedig ist
  Appointment myApp;
  // Das Labeling, das diesen Agent erzeugt hat
  Labeling myLabeling;
  // Der Agent des vorherigen Termins
  LabelingAgent prevAgent;

  void backtrack(){
    // vorherige Instantiierung zuruecknehmen
    lastInstantiation.retract();
    // die verwendeten Werte sind im Labeling bekannt
    Domain used = myLabeling.getUsedValues(this);
    // wenn noch Werte fuer myApp zur Auswahl stehen
    if(! myApp.getDomain().setMinus(used).emptySet()){
       // neue Instantiierung absetzen
       (new AppInstantiation(myApp)).post(acs)
    }else{
    // wenn es der letzte Wert war
      // und eine vorherige Variable in der Ordnung existiert
      if(prevAgent != null){
         // dann verwendete Werte loeschen
```

```
          myLabeling.clearUsedValues(this);
          // und backtracking weiterleiten an vorherigen Agent
          prevAgent.backtrack();
       }else{
       // this ist der erste Agent und
       // es stehen keine Werte mehr zur Auswahl
          gescheitert();
       }
   }
}
```

Die Methode `gescheitert()` *versucht in* $\mathcal{M}$*Plan den unwichtgsten Termin des Tages, an dem der Termin für das Labeling scheiterte, abzusagen, so daß eine Lösung möglich wird. Die Wichtigkeit von Terminen ergibt sich aus den persönlichen Präferenzen der Teilnehmer. Das Absagen wird durch eine Methode umgesetzt, die allen Beteiligten eine entsprechende Nachricht sendet und anschließend das* `Appointment`*-Objekt und alle Constraints über ihm löscht bzw. zurücknimmt. Dadurch sind die Referenzen auf die Constraintvariablen und Constraints gelöscht und der garbage-collector von Java kann die Objekte aus dem Speicher entfernen.*

5.4.2 *local search*

Lokale Suchalgorithmen [Voß00, Nar01, Ste99, Ilog] traversieren im Gegensatz zu klassischen Algorithmen nicht einen implizierten Suchbaum, der einen Suchraum aufspannt, sondern navigieren im Suchraum selbst durch lokale Zustandsveränderungen. Ausgehend von einem zufällig generierten Anfangszustand wird versucht, durch lokale Veränderungen einen Zustand zu erzeugen, der als Lösung akzeptiert werden kann. In der CP werden solche Zustände als Variablenbelegungen erzeugt und analysiert. Bei *constraint satisfaction* kann dann jeweils die Konsistenz mit den Constraints des CSP untersucht werden. Lokale Suchverfahren werden traditionell meist zur Lösung von Optimierungsproblemen eingesetzt, da solche Probleme im *operations research*, in der die meisten *local-search* Algorithmen entwickelt wurden, zu lösen sind. Zur Lösung von Optimierungsproblemen wird eine mehrwertige Bewertungsfunktion verwendet, mit der Zustände verglichen werden können. Bei CSPs besteht der Wertebereich dieser Bewertungsfunktion meist aus einem Wert für Konsistenz und mehreren Werten für Inkonsistenz (z.B. [CD01]). Inkonsistente Variablenbelegungen werden dann durch den Algorithmus solange verbessert, bis sie konsistent sind. Die Verbesserung der Zustände wird durch die Auswahl des bzgl. der Bewertungsfunktion besten Zustands einer sogenannten Nachbarschaft des aktuellen Zustands erreicht. In der Nachbarschaft eines Zustands befinden sich die Zustände, die sich durch Anwendung einer bestimmten Operation aus dem aktuellen Zustand berechnen lassen. In der CP besteht die Nachbarschaft einer Variablenbelegung aus den Variablenbelegungen, die sich an einer bestimmten Menge von Variablen unterscheiden.

Durch die lokalen Veränderungen werden bei *local-search* sehr oft Optima berechnet, in denen keine Verbesserung in der Nachbarschaft möglich ist, die aber nicht als akzeptable Lösung betrachtet werden können. Um die Variablenbelegung in solchen lokalen Optima so zu verändern, daß der Zustand sich wieder auf ein globales Optimum zubewegen kann, muß der Algorithmus ein explizites Verfahren zur Verfügung stellen. Dieses Verfahren sollte auch verhindern, daß nach mehreren Iterationen wieder das gleiche lokale Optimum erreicht wird. Solche Situationen könnten nämlich zu endlosen Programmzyklen führen, weil *local-search*-Algorithmen normalerweise nicht explorativ sind. Bereits betrachtete Zustände werden nicht vollständig erfaßt, man verwendet lediglich in manchen Algorithmen sogenannte Tabu-Listen [GL93], die bestimmte Werte für eine bestimmte Anzahl von Iterationen ausschließen.

Der lokale Suchalgorithmus, den ich für J.CP implementiert habe, ist eine Variante des *simulated annealing* (SA) [JRMS89, JRMS91, DH96]. Der Algorithmus wurde sehr erfolgreich für vielfältige Optimierungsprobleme wie z.B. das *traveling salesman problem* angewandt. Mit der oben genannten Bewertungsfunktion zur Verbesserung inkonsistenter Zustände ist er aber auch für *satisfaction* Probleme geeignet [Wol01]. Die Methode, mit der Variablenbelegungen beim SA aus lokalen Optima "herausnavigiert" werden, wurde vom Prozeß des Abkühlens von Flüssigkeiten bis zu einer kristallinen Form (das *annealing*) abgeleitet. Um beim Abkühlen von Flüssigkeiten einen geordneten kristallinen Zustand mit minimaler Energie zu erreichen, darf die Abkühlung nicht zu schnell erfolgen. Das entstandene Kristall repräsentiert dann ein globales Optimum des Problems bzw. eine Lösung des CSP. Der Algorithmus ist in Abb. 5.6 analog zu [DH96] schematisch dargestellt.

1. Wähle eine zufällig erzeugte initiale Variablenbelegung σ und eine initiale Temperatur T,
2. wiederhole das Folgende n mal,
   (a) wähle eine Variablenbelegung σ' aus der Nachbarschaft von σ.
   (b) Sei *eval* eine Bewertungsfunktion, dann ersetzte σ durch σ', falls $eval(\sigma') < eval(\sigma)$ oder $e^{-(eval(\sigma)-eval(\sigma'))/T} > r$ gilt. Wobei $r \in [0;\ 1]$ zufällig gewählt wird.
3. Verringere T.
4. Wenn noch keine Lösung gefunden wurde oder ein anderes Terminierungskriterium erfüllt ist, dann gehe zu 2.

Abbildung 5.6: Der *simulated annealing* Algorithmus zur lokalen Suche.

Wenn sich im Algorithmus eine Verbesserungsmöglichkeit in der Nachbarschaft eines Zustands ergibt, so wird diese gewählt. Ansonsten kann der aktuelle Zustand auch so verändert werden, daß sich eine Verschlechterung bzgl. der Bewertungsfunktion ergibt, wenn der Wert $e^{-(eval(\sigma)-eval(\sigma'))/T}$ einen gewählten Zufallswert überschreitet. Hierdurch werden lokale Optima verlassen. Durch die Verringerung der Temperatur (Schritt 3.) wird die zufällige Veränderung mit zunehmender Iteration immer unwahrscheinlicher durchgeführt, so daß die Veränderungen des Zustands immer geringfügiger werden. Der Algorithmus terminiert, wenn eine Lösung gefunden wurde oder er anderweitig von außen durch ein Terminierungskriterium (Schritt 4.) beendet wird. Als solches Kriterium wird oft ein *timeout* gewählt, der die Berechnung nach einer gewissen Zeit abbricht.

Dann kann der (oder ein anderer) Suchalgorithmus entweder neu gestartet werden oder es kann angenommen werden, daß keine Lösung existiert.

Wenn, wie beim inkrementellen Constraintlösen, Zustände nicht nur durch Belegung von Variablen und Konsistenzprüfung sondern auch durch Propagation berechnet werden, kann die Nachbarschaft eines Zustands nur schwer vollständig erzeugt werden. Wenn die Belegung einer Variablen verändert wird, so kann dieser Wert durch *forward-checking* Auswirkungen auf andere Variablen haben, die berechnet werden müssen. Um also mehrere benachbarte Zustände zu erzeugen, müßten mehrfach Instantiierungen mit Propagation abgesetzt, bewertet und zurückgenommen werden. Die Bewertungen könnten dann jeweils abgespeichert werden, so daß daraus die Entscheidung in Schritt 2(b) aus Abb. 5.6 getroffen werden könnte. Dieses Vorgehen ist aber wegen der Propagation zu ineffizient um für *local-search* verwendet werden zu können. Daher muß die Nachbarschaft schon bei der Erzeugung eines Zustands implizit berechnet werden [FLL01, PG96]. Aus dieser Nachbarschaft wird dann ein Nachfolgezustand ausgewählt, der eventuell noch gar nicht berechnet wurde, der aber nach einer bestimmten Bewertungsfunktion als optimal betrachtet werden kann. Durch diese a priori Entscheidungen wird der Algorithmus öfter als bei der Untersuchung bestehender Nachbarschaften "falsche" Veränderungen vornehmen. Wenn dies nicht zu häufig vorkommt, so gefährdet es den Erfolg des SA aber nicht, weil hier zur Verhinderung lokaler Optima ohnehin auch Verschlechterungen zugelassen werden.

In der J.CP Implementierung von SA ist die Nachbarschaft eines Zustands durch die Zustände gegeben, die sich durch Änderung der Werte einer festen Zahl von Variablen ergeben. Diese Zahl wird aus der Anzahl der zu instantiierenden Variablen des Suchconstraints berechnet. Lokale Verbesserungen (bzw. Veränderungen um lokalen Optima auszuweichen) werden also durch Rücknahme von Instantiierungen bestimmter Variablen und durch Absetzen neuer Instantiierungen für diese Variablen erreicht. Man gibt also für Schritt 2(b) aus Abb. 5.6 einen Suchalgorithmus im Sinne von Def. 5.1.6 an. Durch Experimente hat sich hierfür als beste Vorgehensweise eine Mischung aus zufälliger Auswahl der Variablen und der Auswahl aufgrund der Analyse aufgetretener Inkonsistenzen ergeben. Bei den Constraints, die durch die aktuelle Variablenbelegung nicht erfüllt waren, wird das Vorkommen jeder Variable gezählt. Die Instantiierungen der Variablen, die hier am häufigsten gezählt wurden, werden dann neben denen einiger zufällig gewählter Variablen zurückgenommen. Es wird also wie beim *limited discrepancy search*- Algorithmus [WDH95, Wal97] versucht, Veränderungen an den Stellen anzubringen, an denen besonders viele Inkonsistenzen auftraten. Durch die Verwaltung einer Tabu-Liste wird, wie beim *tabu search* [GTdW93], dabei verhindert, daß Instantiierungen für einzelne Variablen in unmittelbar aufeinanderfolgenden Iterationsschritten zurückgenommen werden. Mit diesem Vorgehen und der zufälligen Auswahl werden lokale Zyklen bei der Iteration verhindert. Dieses Vorgehen ist allerdings lediglich eine Heuristik zur Vermeidung von Zyklen und lokalen Optima, es ergibt sich daraus kein explorativer Algorithmus.

Die Architektur des SA Suchconstraints entspricht weitgehend der vom chronologischen Backtracking aus Abb. 5.3. Das Labeling stellt einen `EventListener` zur Verfügung, der den des Solvers für die Zeit der Suche ersetzt. Außerdem wird für jede Variable eine Agent gestartet, der Instantiierungen absetzt und zurücknimmt. Die beschriebene Auswahl der zurückzunehmenden Instantiierungen wird im Suchconstraint selbst durchgeführt. Auch die SA Implementierung in J.CP stellt eine Methode zur Berechnung weiterer Lösungen `nextSolution` zur Verfügung, die allerdings nicht explorativ ist, d.h. man kann damit gleiche Lösungen mehrfach erhalten. Im Unterschied zum *backtracking* findet keine Kommunikation zwischen den Agenten statt, sie können asynchron Variablenveränderungen vornehmen, die dann jeweils propagiert werden. Dabei werden im Aufgabenpuffer immer wieder die Propagationsmethoden der Constraints des CSP abgesetzt, die wenn sie ausgeführt werden, Inkonsistenz-Ereignisse erzeugen können. Wenn alle Variablen instantiiert sind und keine Inkonsistenzen mehr registriert wurden, so ist eine Lösung gefunden, weil alle Propagationsmethoden der Constraints über den zu instantiierenden Variablen ausgeführt wurden (Def. 3.3.6.6). Das Suchconstraint bricht die Iteration ebenfalls ab, wenn eine bestimmte vom Benutzer anzugebende Dauer vergangen ist. In diesem Fall wird dann davon ausgegangen, daß keine Lösung existiert.

Die Varianz der Ausführungszeit dieser Suchconstraints ist durch die vielen zufälligen Eingabegrößen sehr groß, es werden aber im Durchschnitt schneller Lösungen gefunden, als mit dem explorativen *backtracking* Constraint aus Abschnitt 5.4.1. Das SA Constraint ist daher vor allem dann zu verwenden, wenn nur eine Lösung möglichst schnell gefunden werden soll. Bei CSPs mit relativ vielen Lösungen bzgl. der Gesamtzahl der Zustände ist der Algorithmus besonders schnell, denn oft sind in solchen Problemen schon die zufällig erzeugten initialen Variablenbelegungen nur an wenigen Stellen inkonsistent. Eine Analyse der Laufzeiten im Bezug zu unterschiedlichen Problemen sowie die Varianz der Laufzeit wird in Abschnitt 7.2 noch genauer dargestellt werden.

Kapitel 6

Verteilungsverfahren

Constraintprobleme können, wie z.B. bei der Terminplanung, verteilt auftreten oder zu komplex sein, um zentral gelöst zu werden. Daher ist die Behandlung verteilter CSPs ein wichtiges Forschungsgebiet der Constraintprogrammierung geworden. In diesem Kapitel werden die besonderen Probleme der Verteilung von Constraintproblemen untersucht und die Eignung von ACS als Ausführungsmodell für solche Anwendungen dargestellt. Beim ACS sind keine zentralen Steuerungsstrukturen notwendig, so daß damit offene, verteilte Systeme gleichartiger Agenten zur Bearbeitung verteilter CSPs direkt implementiert werden können. Durch die inhärente Asynchronität der Constraintausführung ergeben sich durch die Verteilung aus Sicht einzelner Agenten keine Performanzdefizite. Die Implementierung globaler Constraints zur Suche sollte allerdings in verteilten Systemen angepasst werden, damit auch hier möglichst wenig zentrale Steuerung notwendig ist.

Die Implementierung von verteilten Anwendungsprogrammen wird mit der zunehmenden Vernetzung von Computersystemen und der damit erforderlichen Kommunikation in immer mehr Anwendungsgebieten vorangetrieben. Softwaresysteme werden aus folgenden Gründen verteilt und nicht monolithisch, also in einem Prozeß ausführbar, implementiert:

- Die zu lösenden Probleme sind zu komplex, um in annehmbarer Zeit oder im vorhandenen Speicherplatz von einem Rechner gelöst werden zu können.

- Die Probleme treten verteilt, also durch nebenläufige Eingaben an verschiedenen Computern, auf und können aus Gründen der *privacy* oder *security* nicht zentral zusammengefaßt und gelöst werden.

Der erste Punkt ist für das Lösen komplexer Constraintprobleme relevant, wobei allerdings immer zu prüfen ist, ob die Verarbeitung in einem verteilten System wirklich (schneller) erledigt werden kann, als es bei einem zentralen Solver (als *server*) in einer *client-server*-Architektur der Fall wäre. Die zentrale Verarbeitung kann bei großen Probleme auch wegen des Speicherbedarfs unmöglich sein [Sim]. Einerseits steht zwar durch die Verwendung mehrerer Computer mehr Rechenzeit zur Verfügung, andererseits entsteht durch die Verteilung und das verwendete Ausführungsmodell immer zusätzlicher Kommunikationsbedarf. Bei Constraintproblemen mit dichtem Constraintnetzwerk wächst dabei oft der Kommunikationsaufwand schneller als der linear mit der Anzahl von Computern fallende Rechenaufwand pro Rechner. Trotzdem kann in gut organisierten verteilten Anwendungen die gesamte Rechenzeit geringer sein. Daher ist immer eine gute Konfiguration des verteilten Systems zu finden ist, was ausführlich in [Han01] untersucht wurde.

Der zweite Punkt erzwingt dagegen die Implementierung in einem verteilten System. Wenn lokale Daten nicht nach außen gegeben werden sollen (*privacy*) oder aus Sicherheitsgründen nicht durch die vorhandene Kommunikationsinfrastruktur geleitet werden dürfen (*security*), so muß die Anwendung verteilt umgesetzt werden. In meiner Fallstudie zur verteilten Terminplanung gehe ich z.B. davon aus, daß ein Benutzer andere Personen nach "Wichtigkeit" einstufen kann, um Terminen mit diesen Personen unterschiedliche Priorität zu geben. Solche persönlichen Präferenzen sollen aus Gründen der *privacy* nicht nach außen sichtbar sein, so daß sie ausschließlich lokal verarbeitet werden können.

Zur Implementierung verteilter Anwendungen verwende ich das Konzept der Softwarekomponenten [PS96, BW98], indem jeder Agent eine Komponente darstellt. Mehrere solcher Komponenten zusammen ergeben dann ein Softwaresystem zur Bearbeitung verteilter Constraintprobleme. Komponenten sind gekapselte Einheiten, deren Implementierung in einer *black-box* versteckt wird. Das einzige verwendbare Merkmal einer Komponente ist ihre Schnittstelle, durch die sie mit anderen Komponenten kommuniziert. Diese Schnittstelle stellt in der verteilten Multi-Agenten-Architektur auch Methoden zur Verfügung, die von physisch getrennten Computern durch eine gegebene Kommunikationsinfrastruktur durch eine entsprechende Nachricht asynchron aufrufbar sind. Durch diese Architektur wird *privacy* als Grundvoraussetzung angenommen, indem nur die Daten nach außen sichtbar sind, die explizit in der Schnittstelle exportiert werden. *security*-Aspekte können durch eine sichere Implementierung der Schnittstelle und der Datenübermittlung (z.B. durch Verschlüsselung) umgesetzt werden.

Definition 6.0.1 (Komponente) Eine *Komponente* eines Softwaresystems S ist ein abgeschlossene Einheit mit einer eindeutig definierten Schnittstelle. Eine Komponente A heißt *entfernt* von einer Komponente B, wenn A und B keinen gemeinsamen Speicherbereich verwenden. Die Komponenten von S werden in der Menge $\mathsf{components}(S)$ zusammengefaßt.

Fallstudie, Teil 17
Das verteilte Terminplanungstool $\mathcal{M}$Plan besteht aus beliebig vielen gleichartigen Komponenten, deren Schnittstellen Methoden zur Verfügung stellen, die notwendig sind um alle Constraints einhalten zu können. Wie in Abb. 6.1 dargestellt, besteht die Schnittstelle jeder Komponente aus den Klassen der Termine (Appointment) und einer Kommunikationseinheit, die durch entfernte Komponenten erreichbar ist. Die Kommunikation, die vor dem Constraintlöseprozeß stattfinden muß, nämlich das Übermitteln von Kommunikationsdaten, wird durch die allgemeine Kommunikationskomponente erledigt. Das Lösen der Constraints erfolgt danach durch Kommunikation von Constraints und Variablen über diese Schnittstellen. Die persönlichen Präferenzen und der aktuelle Terminkalender der Benutzer sind von entfernten Komponenten aus nicht sichbar, da diese Komponenten innerhalb der Komponente (balck-box) gekapselt sind. Dadurch ist

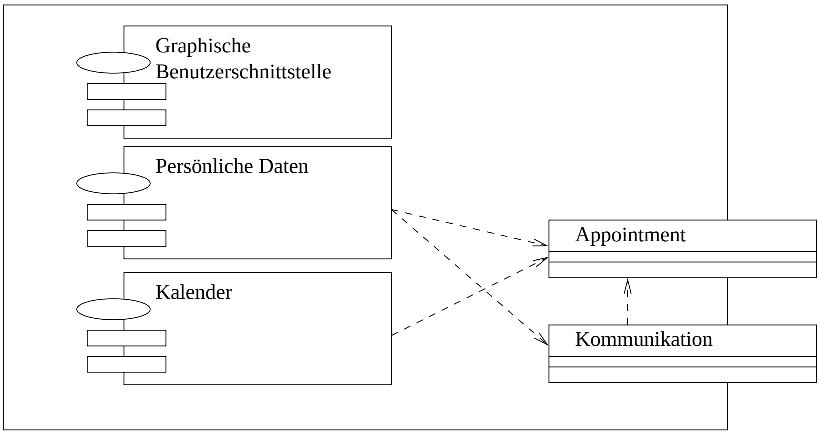

Abbildung 6.1: Die Klassen und Komponenten eines Agenten in $\mathcal{M}$Plan-System. Mehrere dieser Agenten werden als Komponenten eines verteilten Systems über die angegebenen Schnittstellen gekoppelt.

privacy für die persönlichen Daten, wie z.B. Präferenzen von Personen und für bereits bestehende Termine grundsätzlich gesichert. Nach außen ist lediglich die Information darüber sichtbar, die explizit durch die Schnittstellen zur Verfügung gestellt wird.

Unterschiedliche Agenten bzw. Komponenten in einem Softwaresystem kommunizieren durch Nachrichten. Diese sind (in meinem Ansatz) Methodenaufrufe in entfernten Komponenten, bei denen als Parameter auch (sequentialisierbare) Objekte übermittelt werden können. In der Regel veranlassen Nachrichten damit nur, daß eine Methode ausgeführt wird, ohne deren Terminierung abzuwarten. Nur wenn dies explizit verlangt wird oder die Mehtode einen Rückgabewert liefert, muß der Aufruf synchron erfolgen. Durch die physische Trennung bzw. die Ausführung in separaten Prozessen von Nachrichtensender und -empfänger unterscheidet sich ein solcher Methodenaufruf durch seine Umsetzung und damit u.a. durch seine Ausführungszeit von Methodenaufrufen innerhalb eines Prozesses. Mit der Implementierung von Nachrichten und deren Übermittlung befasse ich mich in dieser Arbeit aber nicht und setzte wie [YD98] voraus, daß jede Nachricht irgendwann ankommt. Die technischen Voraussetzungen für diese Annahme werden z.B. in [FC98, FCar] ausführlich beschrieben.

Definition 6.0.2 (Nachricht) Eine *Nachricht* ist ein (asynchroner oder explizit synchronisierter) Methodenaufruf in der Schnittstelle einer entfernten Komponente.

Der mehrfache Austausch von Nachtrichten zwischen entfernten Komponenten wird als Kommunikation bezeichnet (z.B. [Han01]). Durch die Asynchronität der Methodenaufrufe, die durch die Implementierung als Nachricht entsteht, kann die Kommunikation zwischen Komponenten beliebig lange dauern. Da ich lediglich voraussetze, daß jede Nachricht irgendwann ankommt, kann die Kommunikation zwischen Komponenten z.B. bei einem Kabelbruch solange dauern, bis das Kabel repariert wurde. Da die Ausführung der Propagationsmethoden für die Constraintverarbeitung und insbesondere der Terminierung der Constraintpropagation relevant ist (vgl. Lem. 3.4.3, Lem. 3.5.4), definiere ich den Begriff der abgeschlossenen Kommunikation.

Definition 6.0.3 ((abgeschlossene) Kommunikation) Das Verschicken von Nachrichten zwischen Agenten heißt *Kommunikation*. Die Kommunikation in einer Menge paarweise entfernter Komponenten A heißt *abgeschlossen*, wenn jeder Methodenaufruf in den Schnittstellen aller $a \in A$ ausgeführt wurde.

6.1 Verteilte Constraint *satisfaction*

Zusammenfassung. Durch die Implementierung constraintbasierter Programme in verteilten Softwaresystemen können verteilte Constraintprobleme bearbeitet werden. Diese Probleme treten entweder natürlich verteilt auf oder werden künstlich verteilt, weil sie wegen technischer Beschränkungen nicht in monolithischen Anwendungen gelöst werden können. In diesem Abschnitt wird das Konzept der Verteilung durch Agenten vorgestellt, die jeweils für eine bestimmte Menge von Constraints und Variablen zuständig sind.

Ein *verteiltes CSP* (*Distributed CSP:* DCSP) ist gegeben durch ein CSP, dessen Variablen und Constraints über verschiedene entfernte Komponenten verteilt sind. Dabei kommunizieren die Komponenten nach dem folgenden Modell:

- Komponenten kommunizieren durch Nachrichtenaustausch. Eine Komponente kann einer anderen eine Nachricht schicken, wenn sie deren Adresse kennt.
- Die Dauer, in der eine Nachricht verschickt wird, ist unvorhersehbar aber endlich.

Die Implementierung von Constraintprogrammen als Erweiterung beliebiger (imperativer) Programme (Def. 3.3.1) erlaubt die Verwendung des Verteilungsmodells der jeweils gewählten Programmiersprache in constraintbasierter Software. Für die Implementierungssprache Java meiner prototypischen ACS-Implementierung J.CP existieren mehrere Konzepte und Pakete zur Umsetzung verteilter Systeme. Je nach Anwendungsbereich des Constraintprogramms können unterschiedliche Pakete wie Java RMI [Oeb01], Enterprise JavaBeans [Rom99] oder JINI [AO99] verwendet werden. In J.CP wird Verteilung mit Java RMI direkt unterstützt, was in Abschnitt 6.3 genauer beschrieben wird. All diese Verteilungskonzepte gehen von einer Komponenten-basierten Verteilung der Objekte aus. In jedem verteilten Constraintprogramm werden Komponenten mit klar definierten Schnittstellen zur Kommunikation definiert. Im Speicherbereich jeder dieser Komponenten befinden sich eindeutig bestimmte Constraints und Constraintvariablen. Die einzelnen Objekte kommen in der verteilten Anwendung nur genau einmal vor und es existiert, im Unterschied zu Verteilungskonzepten mit innerem und äußerem Constraintlösen [Han01], zu jeder (konzeptionellen) Variable nur genau ein (implementiertes) Variablen-Objekt. Die Handhabung der Variable durch entfernte Objekte wird zwar durch weitere Kommunikationsobjekte unterstützt, aber es wird keine Kopie der Variable in dem entfernten System angelegt. Die Verteilung dieser Objekte in einer Anwendung wird als Konfiguration bezeichnet.

Definition 6.1.1 (Konfiguration) Gegeben sei ein Constraintprogramm P, dann ist eine Äquivalenzrelation in $\mathsf{vars}(P) \cup \mathsf{constraints}(P) \cup \mathsf{solvers}(P)$ eine *Konfiguration* von P.

Durch die Definition von Konfigurationen mit einer Äquivalenzrelation entsteht eine Partition auf allen Constraints, Variablen und Solvern. Jede Äquivalenzklasse enthält eine Teilmenge aller vorhandenen Objekte, so daß jedes Objekt genau einer Äquivalenzklasse zugeordnet werden kann. Diese Partition wird dann in konkreten Anwendungen durch die Verteilung so umgesetzt, daß die Objekte jeder Äquivalenzklasse in genau einer Komponente des Programms enthalten sind. Solche gekapselten Komponenten, die eine bestimmte Teilmenge aller Constraints und Constraintvariablen verwalten, werden in der verteilten Constraintprogrammierung als Agenten bezeichnet [Han01, YD98]. In meiner objekt-orientierten Implementierung bedeutet dies, daß die jeweiligen Objekte fest in dem Speicher des Agenten stehen, in dem sie erzeugt wurden. Ein Verschieben von Objekten zwischen verschiedenen Komponenten oder das Verschieben von Komponenten zwischen physisch getrennten *Hardware*-Einheiten (*mobile-agents*) wird durch meinen Ansatz nicht unterstützt.

Definition 6.1.2 (Agent) Gegeben sei ein Constraintprogramm P und eine Konfiguration K, dann heißt jede Komponente P' von P ein *Agent* von P, falls gilt, daß $\mathsf{vars}(P') \cup \mathsf{constraints}(P') \cup \mathsf{solvers}(P')$ eine Äquivalenzklasse aus $P/_K$ ist.

Bemerkung 6.1.1
Ein Constraintprogramm mit mehreren Agenten wird verteiltes Constraintprogramm genannt, wenn die Agenten keinen gemeinsamen Speicherbereich verwenden.

Im ACS Ausführungsmodell ist es bei der Propagation notwendig, daß in jeder Variable bekannt ist, in welcher Komponente sie enthalten ist. Daher gebe ich eine totale Abbildung an, die die Zugehörigkeit von Objekten zu Agenten definiert. Diese Abbildung kann lokal in den Objekten durch Referenzen auf ihre Agenten implementiert werden.

Definition 6.1.3 (Zugehörigkeit) In einem verteilten Constraintprogramm P ist die *Zugehörigkeit* definiert durch die (totale) Abbildung $\mathsf{belongs} : \mathsf{vars}(P) \cup \mathsf{constraints}(P) \cup \mathsf{solvers}(P) \to \mathsf{components}(P)$, die jedem Objekt seinen Agenten zuordnet.

Fallstudie, Teil 18
Die Terminplanungsprobleme, die mit $\mathcal{M}$Plan gelöst werden können, werden in einem offenen, verteilten System definiert. In den unterschiedlichen Agenten werden Variablen deklariert und Constraints darüber und über entfernte Variablen abgesetzt. Daraus entsteht ein verteiltes CSP, in dem bei $\mathcal{M}$Plan in jedem Agent genau ein Solver benutzt wird. Alle Constraints benutzen den gleichen Speicher wie ihr Solver und die Kommunikation wird durch Zugriff auf die Variablen durchgeführt. Ein typisches Szenario ist als Constraintnetz in Abb. 6.2 angegeben.

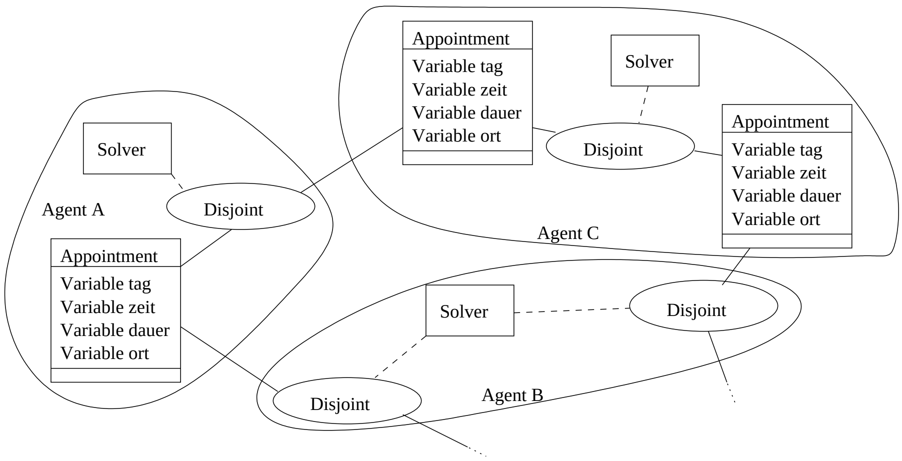

Abbildung 6.2: Ein Constraintnetz bei der Terminplanung mit $\mathcal{M}$Plan, die Variablen und Constraints sind über mehrere Agenten verteilt. In jedem Agent läuft genau ein Solver.

6.2 Verteilung in ACS

Zusammenfassung. In diesem Abschnitt werden die Schnittstellen von Agenten zur Constraintverarbeitung mit ACS in verteilten Umgebungen definiert. Die Architektur verteilter Anwendungen und die Möglichkeiten der Konfigurationen werden angegeben und verifiziert. Dazu wird gezeigt, daß das ACS Ausführungsmodell ausschließlich die Methoden der angegebenen Schnittstellen verwendet. Die Nachrichten können in ACS durch nichtblockierende Kanäle verschickt werden, weil die dadurch entstehende Asynchronität durch das Ausführungsmodell berücksichtigt wird. Damit ist, im Gegensatz zu allen existierenden Systemen zur verteilten Constraintverarbeitung, keine Einschränkung der Kommunikation zwischen den einzelnen Objekten im Vergleich zu monolithischen Anwendungen nötig. Schließlich wird gezeigt, daß auch verteilte Implementierungen von ACS die intendierte denotationale Semantik berechnen, wenn jede Nachricht angekommen ist.

Das ACS Ausführungsmodell zur Constraintverarbeitung ist von vornherein so entworfen, daß es ohne zentrale Datenstrukturen auskommt und somit in verteilten Anwendungen direkt einsetzbar ist [Rin01b, Rin01a, RS02]. Bei ACS sind keinerlei Metastrukturen, die die Constraintverarbeitung in separaten Agenten steuern, notwendig. ACS erfordert daher auch keine Unterscheidung von innerem und äußerem Constraintlösen wie in [Han01]. Inneres Constraintlösen innerhalb eines Agenten ist Constraintverarbeitung unter Verwendung eines herkömmlichen monolithischen Solvers wie SICStus [SICS]. Die Ergebnisse solcher lokalen Solver werden dann durch äußeres Constraintlösen synchronisiert und aufeinander abgestimmt. Das Problem dieses Ansatzes besteht darin, die Eigenschaften der lokalen Solver (z.B. Vollständigkeit) durch das äußere Constraintlösen so zu erweitern, daß sie in verteilten Systemen erhalten bleiben.

Die Ausführungsmodelle zur inneren Constraintverarbeitung können in fast allen Fällen nicht für verteilte Anwendungen verwendet werden, weil sie zentrale Datenstrukturen wie z.B. Constraintspeicher aller bekannten Constraints oder Verwaltung ehemaliger globaler Zustände erfordern (vgl. Abschnitt 2.3). Durch eine strenge Trennung von Propagation und Suche kann wie in Mozart [MOz] der Suchraum gekapselt werden, so daß keine globalen Strukturen mehr notwendig sind. Allerdings erfordert auch dieses Ausführungsmodell eine Synchronisierung der Constraints und Suchalgorithmen.

Im Unterschied zu verteilten Suchalgorithmen [YD98] kann durch die inkrementelle Verarbeitung von Constraints in ACS der Suchraum wie bei der CLP a priori durch Propagation eingeschränkt werden. Außerdem sind Konfigurationen bzgl. Variablen und Constraints frei definierbar, so daß diese Objekte in verteilt auftretenden Constraintproblemen immer direkt in dem Agenten erzeugt werden können, in dem sie auftreten. Im Unterschied zu [YD98] können in jedem Agenten beliebig viele Variablen und Constraints gehalten werden, so daß Kommunikation nur durchgeführt wird, wenn es durch die Verteilung des Problems notwendig ist.

Aus der Definition des ACS-Ausführungsmodells in Kapitel 3 kann die notwendige Kommunikation zwischen den einzelnen Objekten abgeleitet werden. Ich nehme an, daß in jedem Agenten bei verteilten ACS Implementierungen genau ein Solver-Objekt gestartet wurde und daß jedes Constraint in dem Solver abgesetzt wurde, der in seinem Prozeß läuft. Dies ist keine Einschränkung der Allgemeinheit, weil jedes Constraint ohnehin genau einem Solver fest zugeordnet wird (Def. 3.3.1.4). Dadurch muß die Interaktion zwischen Solver und Constraint nicht durch Nachrichtenaustausch implementiert werden. Bei der folgenden Definition gebe ich also nur die Methodenaufrufe des ACS Ausführungsmodells (Def. 3.3.1, 3.3.5, 3.5.3) an, bei denen in einer Implementierung als Nachricht, Sender und Empfänger zu unterschiedlichen Agenten gehören können.

Definition 6.2.1 (ACS-Komponente) Eine *ACS-Komponente* ist eine Softwarekomponente, deren Schnittstelle mindestens folgende Methoden zur Verfügung stellt:

1. Definition von Variablendomänen: `entail = setDomain(v,`$\overline{p}$`)`, wobei `v` eine Constraintvariable und $\overline{p}$ eine Domänenbeschreibung ist
2. Domänenanfragen: $\overline{p}$ `= getDomain(v)` wobei `v` eine Constraintvariable ist
3. Zuverlässigkeitsanfragen: `r = getReliability(s)`, wobei `s` ein Constraintlöser ist
4. Constraints zurücknehmen: `retract(c)`
5. Constraints erneut ausführen: `post(c)`
6. Empfangen von Inkonsistenz-Ereignissen:
   `event = receiveInconsistency()`
7. Abhängige Constraints eintragen: `addDepending(v,c,p)`, wobei `v` eine Variable, `c` das einzutragende Constraint und `p` ein Merkmal der Domäne von `v` ist.
8. Konsequenzen eines Constraints `c` berechnen: C' = `getConsequences(c,C)`

9. Verwendende Constraints eines Constraints `c` berechnen:
   C' = `getAllLaterPostings(c,C)`

Damit mehrere solcher Komponenten gemeinsam als Constraintprogramm so funktionieren, wie es durch die denotationale Semantik der abgesetzten Constraints zu erwarten ist, müssen die Komponenten intern so implementiert sein, daß die Methodenaufrufe in der Schnittstelle analog zu ACS umgesetzt werden. Ich gehe, wie beschrieben, davon aus, daß in jedem Agent genau ein Solver und ausschließlich Constraints dieses Solvers existieren. Außerdem sind eine beliebige Menge von Variablen enthalten, deren Zughörigkeit eindeutig definiert ist. Die Methodenaufrufe werden dann immer in dem Objekt umgesetzt, dessen Referenz als Argument mit der Nachricht verschickt wurde.

Definition 6.2.2 (ACS-Agent) Ein *ACS-Agent* A ist eine ACS-Komponente mit $\mathsf{solvers}(A) = \{acs\}$ und $\forall(\overline{v}, T, s, z, E, \overline{h}) \in \mathsf{constraints}(A) : s = acs$ für die gilt:

1. $\forall v = (\overline{p}, C) \in \mathsf{vars}(A) : A.\mathtt{setDomain}(v, \overline{p}) = v.\mathtt{setDomain}(\overline{p})$

2. $\forall v = (\overline{p}, C) \in \mathsf{vars}(A) : A.\mathtt{getDomain}(v) = v.\mathtt{getDomain}()$

3. $A.\mathtt{getReliability}() = acs.\mathtt{getReliability}()$

4. $\forall c \in \mathsf{constraints}(A) : A.\mathtt{retract}(c) = c.retract()$

5. $\forall c \in \mathsf{constraints}(A) : A.\mathtt{post}(c) = c.post(acs)$

6. $A.\mathtt{receive}(\mathtt{inconsistencyEvent}) \Rightarrow z = inconsistent$ für $acs = (z, B, \rho)$.

7. $\forall v = (\overline{p}, \{C1, ..., C_p, ..., C_n\}) \in \mathsf{vars}(A) : A.\mathtt{addDepending}(v, c, p) \Rightarrow (v = (\overline{p}, \{C1, ..., \langle C_p, c\rangle, ..., C_n\}))$

8. $\forall c \in \mathsf{constraints}(A) : A.\mathtt{getConsequences}(c, C) = c.getConsequences(C)$

9. $\forall c \in \mathsf{constraints}(A) :$
   $A.\mathtt{getAllLaterPostings}(c, C) = c.getAllLaterPostings(C)$

Aufrufe der Methoden der Schnittstelle von A mit anderen Argumenten sind undefiniert.

Bemerkung 6.2.1
Zu jedem Agent gehört genau ein Solver und ausschließlich Constraints dieses Solvers.

Wird in einem ACS-Agent eine Methode der Schnittstelle aufgerufen, in der eine Referenz auf ein in diesem Agenten unbekanntes Objekt übergeben wird, so wird dies als fehlerhafter Aufruf betrachtet. Solche Fehler können in ACS einfach verhindert werden, wenn bei jeder existierenden Referenz auf ein entferntes Objekt die Zugehörigkeit eindeutig definiert ist. Dies wird in J.CP durch die Verwendung von *stub*-Klassen umgesetzt, die als *remote*-Objekte implementiert werden, in denen neben einigen statischen Eigenschaften der jeweiligen Objekte auch die nötigen Informationen zur Kommunikation mit dem entfernten Objekt enthalten sind (vgl. Abschnitt 6.3) . Mit diesen *stub*-Objekten werden entfernte Objekte in den einzelnen Agenten referenziert. Diese lokalen Referenz-Objekte "delegieren" die Methodenaufrufe durch entsprechende Nachrichten an die eigentlichen Objekte. In einem lokalen *stub*-Objekt vs einer Variable v wird z.B. der Methodenaufruf $v.getDomain()$ durch die Nachricht $vs.getAgent().getDomain(v)$ implementiert. Dabei definiert die Methode $getAgent$ die Zugehörigkeitsabbildung aus Def. 6.1.3 punktweise, d.h. in jedem Objekt ist die Referenz auf ihren Agenten definiert. Durch diese "automatische Übersetzung" des lokalen Methodenaufrufs in eine Nachricht an den richtigen Adressaten und die feste Bindung der Objekte an ihre Agenten können keine fehlerhaften Aufrufe in falschen Agenten auftreten.

Fallstudie, Teil 19
In $\mathcal{M}$Plan wird die Java-RMI Technik zur Kommunikation der Komponenten verwendet (vgl. Abschnitt 6.3). Hierzu werden Java interface-Klassen implementiert, die alle Methoden exportieren, die via Internet aufrufbar sind. In Abb. 6.3 werden die für $\mathcal{M}$Plan implementierten und die von $\mathcal{M}$Plan verwendeten abstrakten J.CP-Klassen dargestellt. Die notwendigen Methoden zur Kommuni-

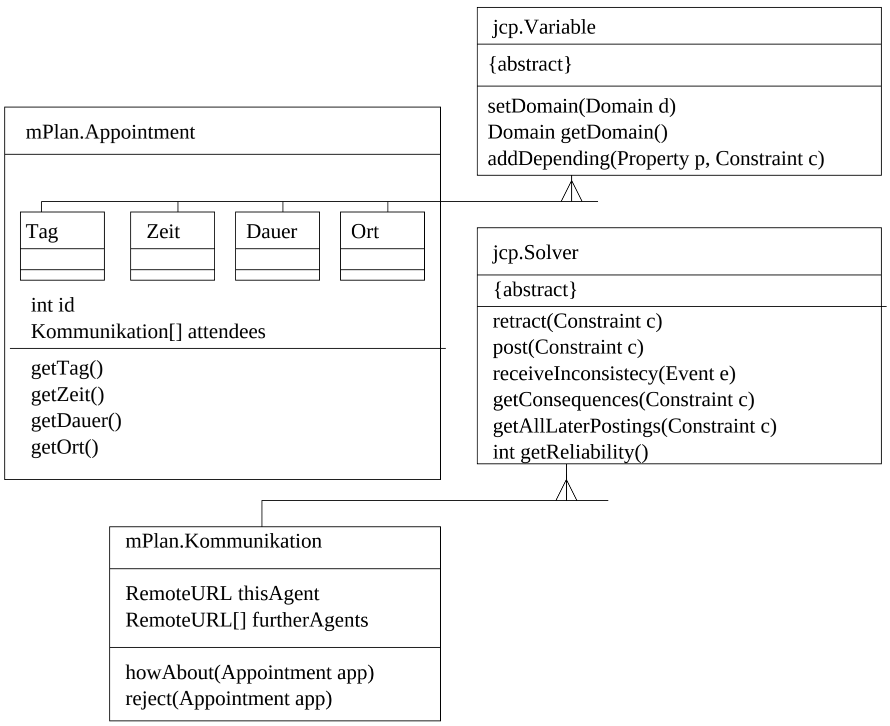

Abbildung 6.3: Die *interface*-Klassen des Terminplaners $\mathcal{M}$Plan, die für den *remote*-Zugriff mit Java RMI exportiert werden.

kation mit den Variablen werden durch die J.CP interface-Klassen 'Variable' und 'Solver' exportiert. Durch die $\mathcal{M}$Plan interface-Klassen 'Appointment' und 'Kommunikation' werden vor Beginn der Constraintverarbeitung alle notwendigen Daten ausgetauscht. Die Kommunikationseinheit stellt die Methode 'howAbout(Appointment a)' zur Verfügung, mit der dem Agenten ein Termin vorgeschlagen werden kann. Mit der Methode 'reject(Appointment a)' werden Termine *abgesagt, die dieser Agent vorgeschlagen hat. Mit dem Aufruf von 'howAbout' wird im Parameter auch die Referenz auf die relevanten Constraintvariablen und einige andere Daten übermittelt, die später zur Festlegung des Termins benötigt werden.*

Mehrere ACS-Agenten zusammen bilden ein verteiltes Constraintprogramm, wenn alle referenzierten Variablen, Constraints und Solver in einem Agenten zur Verfügung stehen.

Definition 6.2.3 (Verteiltes ACS Programm) Eine Menge von ACS-Agenten A werden als *verteiltes ACS-Programm* P bezeichnet, wenn für jedes Constraint $(\overline{v}, T, s, z, E, \overline{h}) \in \bigcup_{a \in A} \mathsf{constraints}(a)$ gilt:

1. $\exists a \in A : s \in \mathsf{solvers}(a)$
2. $\forall v \in \varepsilon(\overline{v}) : \exists a \in A : v \in \mathsf{vars}(a)$
3. $\forall c' \in E : \exists a \in A : c' \in \mathsf{constraints}(a)$

und wenn für jede Variable gilt:
$\forall(\overline{p}, \{C_1, ..., C_n\}) \in \bigcup_{a \in A} \mathsf{vars}(a) : \forall c \in \bigcup_{1 \leq i \leq n} \varepsilon(C_i) : \exists a \in A : c' \in \mathsf{constraints}(a)$.
Die Menge aller Agenten in P wird dann zusammengefaßt als $\mathsf{agents}(P)$.

Jede ACS-Komponente sollte als eigenständiges ausführbares Programm implementiert werden, so daß sich die einzelnen Agenten getrennt starten und beenden lassen. Nur so können verteilte Anwendungen so implementiert werden, daß sie durch immer neue Agenten erweitert werden. Mit jedem neu gestarteten Agent können neue Variablen und Constraints in das implementierte Constraintnetzwerk eingefügt werden. Bevor ein Agent beendet wird, sollte er beachten, daß auch wenn seine lokalen Variablen und Constraints nicht mehr existieren, ein abgeschlossenes Constraintprogramm erhalten bleibt. Es muß also sichergestellt werden, daß keine Nachrichten an nicht mehr existierende Agenten geschickt werden. Wenn ein Agent bei seiner Beendigung noch Objekte enthält, die von anderen Agenten referenziert werden, so müssen diese bei seinem Neustart wieder zur Verfügung gestellt werden und alle neu hinzugekommenen Nachrichten bearbeitet werden. Ist dies nicht gesichert oder wird der Agent nicht mehr gestartet, so entspricht die Implementierung nicht meinem in Abschnitt 6.1 definiertem Kommunikationsmodell. Ich gehe also bei der Ausführung verteilter Constraintprogramme davon aus, daß jede Referenz existiert oder, die an sie gerichteten Nachrichten nach endlicher Zeit verarbeitet werden. Die einzelnen Agenten terminieren per se nicht, weil sie ja jeweils einen unendlichen Solverthread (Abb. 3.3) beinhalten.

Definition 6.2.4 (Ausführung verteilter ACS Programme) Ein verteiltes ACS Programm wird ausgeführt, indem die Implementierungen all seiner Agenten ausgeführt werden.

Fallstudie, Teil 20
Die einzelnen Agenten im $\mathcal{M}$Plan-System stellen all ihre angebotenen Referenzen solange zur Verfügung, wie sie von anderen Agenten benötigt werden könnten. Insbesondere muß auch auf Nachrichten reagiert werden, die eintrafen, solange der Agent nicht durch die gegebene Kommunikationsinfrastruktur

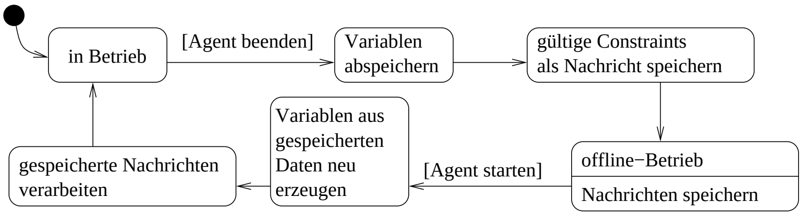

Abbildung 6.4: Statechart eines Agenten in $\mathcal{M}$Plan. Nach seiner initialen Erzeugung verarbeitet er alle ankommenden Nachrichten. Beim Beenden seines Prozesses werden alle Variablen gesichert und ein *mail-daemon* kontaktiert, der ankommende Nachrichten auf einem laufenden Rechner zwischenspeichert. Vorher werden alle noch gültigen lokalen Constraints ebenfalls als Nachricht gespeichert, um beim Neustart wiederhergestellt und abgesetzt zu werden. Sobald der Prozeß wieder gestartet wird, werden all diese Nachrichten dann verarbeitet. Diese Funktionalität wird erst in zukünftigen Versionen von $\mathcal{M}$Plan umgesetzt.

erreichbar war. In Abb. 6.4 sind die Zustände eines Agenten in einem statechart [HG96] dargestellt.
Während ihres Betriebs verwalten die einzelnen Agenten in $\mathcal{M}$Plan bekannte 'Appointment'-Objekte in drei verschiedenen Klassen:

1. *Terminanfragen, die der Benutzer noch nicht betrachtet hat*
2. *Terminanfragen, die betrachtet wurden, deren Constraintvariablen aber noch nicht instantiiert sind und der Termin daher noch nicht feststeht*
3. *Termine, die festgelegt wurden. Diese sind im aktuellen Kalender eingetragen.*

Um einen Termin festzulegen, muß er, falls er von jemandem anders vorgeschlagen wurde und stattfinden soll, einmal Element jeder der drei Klassen gewesen sein, also drei Phasen durchlaufen. Wenn er vom Benutzer des Agenten selbst vorgeschlagen wurde, wird Phase 1. übersprungen. Die Zustandsübergänge werden in Abb. 6.5 als statechart eines Agenten dargestellt. Mehrere solcher Agenten können dann als nebenläufige Zustände modelliert werden und ergeben insgesamt ein $\mathcal{M}$Plan Anwendungsprogramm.

Bei der Constraintverarbeitung mit ACS in verteilten Systemen entsteht durch die unbestimmte Dauer in der Nachrichten verschickt werden eine weitere Asynchronität. Asynchrone Nachrichten werden über nicht-blockierende Nachrichtenkanäle verschickt, so daß die Methodenaufrufe in entfernten Objekten durch das Ablegen der Nachricht lokal implementiert werden können. Dadurch können auch Methodenenaufrufe in entfernten Objekten aus der Perspektive des aufrufenden Agenten direkt und ohne unvorhersehbaren Kommunikationsaufwand durchgeführt werden. Wann die Nachricht ankommt und wann die entsprechende Methode dann wirklich ausgeführt wird, ist dem Sender nicht bekannt und in ACS auch nicht relevant. Dadurch ist im Unterschied zur Implementierung verteilter Anwendungen in Mozart [MOz] oder anderer Systeme,

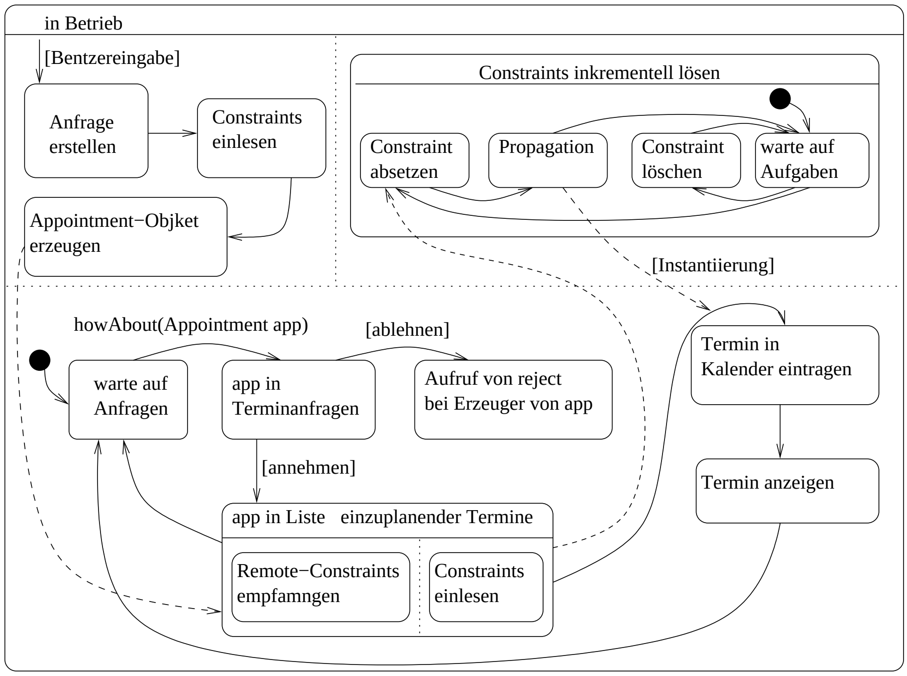

Abbildung 6.5: Die Zustände während des Betriebs eines Agenten in $\mathcal{M}$Plan. Der Constraintlöser verarbeitet nebenläufig die Constraints, die vom lokalen Anwender und von den Benutzern anderer $\mathcal{M}$Plan-Agenten abgesetzt wurden.

die die synchrone Ausführung von Propagationsmethoden erwarten, wie z.B. [RM01], auch *look-ahead*-Propagation über Prozeßgrenzen hinweg in vertretbarer Zeit möglich. In ACS kann diese starke Propagation auch in verteilten Anwendungen genutzt werden, weil sie effektiv keine zusätzliche Asynchronität in das System einbringt. Bei der Propagation werden Constraints erneut abgesetzt, deren Propagationsmethoden aufgrund neuer Domäneneinschränkungen evtl. weitere Einschränkungen in anderen Variablendomänen ableiten können. Da die Ausführung dieser Methoden in ACS ohnehin asynchron zum erneuten Absetzen des Constraints durchgeführt wird, entsteht durch den nicht-blockierenden Nachrichtenaustausch an dieser Stelle lediglich eine weitere Verzögerung bei der asynchronen Ausführung der Methode. Diese "zweite Asynchronität" kann aber genau wie die, die auch in monolithischen Anwendungen auftritt, behandelt werden, weil sie an der gleichen Stelle entsteht. Dadurch braucht das ACS Ausführungsmodell nicht verändert zu werden, um mit nicht-blockierenden Nachrichten verteilte Propagation durchführen zu können.

Fallstudie, Teil 21

Anforderungen an Termine werden in $\mathcal{M}$Plan von beteiligten Agenten in Form von Constraints über Variablen abgesetzt. Die Kommunikation zwischen Constraint und Variable kann dabei Methodenaufrufe in entfernten Komponenten erfordern (vgl. Teil 18 der Fallstudie). Hierdurch entsteht eine Asynchronität beim Lösen des DCSP, die durch das ACS Ausführungsmodell abgefangen wird.

Im Sequenzdiagramm [Boo94] in Abb. 6.6 wird an einem Szenario gezeigt, daß der asynchrone Nachrichtenaustausch an den gleichen Stellen stattfindet wie die asynchrone Constraintverarbeitung. Der in der Methode propagate(), die asynchron aufgerufen wird, wird eine asynchrone Nachricht verschickt, die sonst direkt als Methodenaufruf implementiert werden würde. Die "doppelte" Asynchronität braucht nicht weitergehend als die "einfache" Synchronität von ACS berücksichtigt zu werden.

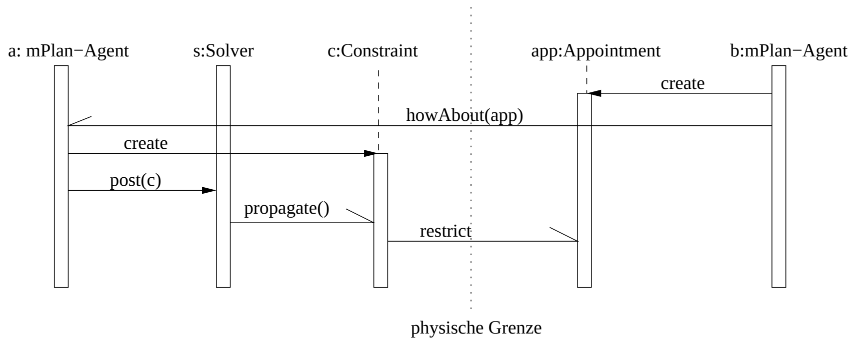

Abbildung 6.6: Ein Sequenzdiagramm der Constraintverarbeitung in einer verteilten Anwendung von $\mathcal{M}$Plan. Synchrone Methodenaufrufe sind durch gefüllte Pfeilspitzen, asynchrone durch halbe Spitzen dargestellt. Durch den asynchronen Aufruf von 'propagate' erfolgt der asynchrone *remote*-Aufruf von 'restrict', der die Variablendomänen von 'app' einschränkt. In monolithischen Anwendungen ist der Aufruf von 'restrict' synchron, was aber keinen Unterschied macht, da er auch dann innerhalb der asynchronen Ausführung von 'propagate' erfolgt.

Um auch theoretisch die Ergebnisse eines verteilten Constraintprogramms mit ACS zu evaluieren, konstruiere ich zu jedem verteilten Programm eine monolithische Implementierung. Um die Eignung von ACS und meiner Definition von ACS-Agenten zur Implementierung verteilter constraintbasierter Systeme zu zeigen, beweise ich, daß die "monolithische Version" einer verteilten Umsetzung eines CSP genau das gleiche Ergebnis liefert, wie die verteilte Implementierung. Zusammen mit Satz 2 ergibt sich daraus auch die Korrektheit und Vollständigkeit verteilter Constraintprogramme bezüglich der (monolithischen) denotationalen Semantik aus Def.3.4.1.

Definition 6.2.5 (Zentralisierung) Gegeben sei ein verteiltes ACS Programm P, dann heißt ein Constraintprogramm P' *Zentralisierung* von P, wenn gilt:

1. $\bigcup_{a \in \mathsf{agents}(P)} \mathsf{constraints}(a) = \mathsf{constraints}(P')$
2. $(\texttt{v = new Variable}(\overline{p})) \in \mathsf{instr}(\bigcup_{a \in \mathsf{agents}(P)} a)$
   $\Rightarrow (\texttt{v = new Variable}(\overline{p})) \in \mathsf{instr}(P')$

Bemerkung 6.2.2
Die Implikation bei 2. reicht aus, weil weitere Variablen wegen 1. ohnehin nicht von den Constraints berührt werden.

Die einzige Einschränkung, die sich bzgl. der Korrektheit der Propagation in verteilten ACS Anwendungen ergibt, folgt aus dem gewählten Kommunikationsmodell. Wie es bei der Untersuchung verteilter Propagation üblich ist [Zoe02], setze ich voraus, daß jeder abgesetzte Methodenaufruf zur Ausführung gekommen ist. Mit der Sicherung dieser Voraussetzung befaßt sich z.B. [FCar].

Lemma 6.2.1 Das Ergebnis eines verteilten ACS-Programms entspricht dem seiner Zentralisierung, wenn die Kommunikation abgeschlossen ist.

Beweis:
Um das Lemma zu beweisen muß gezeigt werden, daß das ACS Ausführungsmodell keine Methoden verwendet, die nicht durch die Schnittstelle eines ACS-Agenten zur Verfügung gestellt werden. Außerdem muß gezeigt werden, daß die Methoden im Endeffekt das gleiche Ergebnis liefern. Letzteres ergibt sich daraus, daß das Ergebnis laut Satz 2 und der Theorie der chaotischen Iteration [Apt98] durch Propagation berechnet wird. Eventuelle Verzögerungen bei der Ausführung der Methoden haben keine Auswirkungen auf das Ergebnis, weil bei dessen Berechnung die Reihenfolge ohnehin unerheblich ist. Ich zeige nun, daß alle notwendigen Methoden durch die ACS-Komponente, also durch die Schnittstelle der Agenten, zur Verfügung gestellt werden:

Monolithisch	Verteilt
Erzeugen von Solvern, Constraints und Variablen, Def. 3.3.4.1-3	wird lokal ausgeführt, es können keine Objekte in entfernten Komponenten angelegt werden
Constraints absetzen und zurücknehmen, Def. 3.3.4.4 und 5, sowie Def. 3.3.6.2 und 3	Wird bei Bedarf delegiert laut Def. 6.2.2.4 und 5, das Eintragen der Abhängigen Constraints (gemäß Def. 3.3.6.6) erfolgt hier explizit durch die Methode `addDepending` aus Def. 6.2.2.7
Lesen und Schreiben von Variablendomänen, Def. 3.3.4.6 sowie Def. 3.3.6.5 und 6	Wird bei Bedarf delegiert laut Def. 6.2.2.1 und 2
Lesen der Zuverlässigkeit, Def. 3.3.4.7	wird delegiert laut Def. 6.2.2.3 oder verteilt berechnet wie in Def. 6.4.1
Aufruf der Arbeitsaufgaben durch den Solver, Abb. 3.3	kann lokal ausgeführt werden, weil laut Def.6.2.2 ausschließlich Constraints c implementiert sind, für die $\mathsf{belongs}(s) = \mathsf{belongs}(c)$ gilt
Ausführung der Constraintrücknahme, Def. 3.5.4 bzw. Abb. 3.9 mit der Berechnung der betroffenen Constraints aus Abb. 3.6 und Abb. 3.7	verwendet zur Berechnung der betroffenen Constraints die Methoden aus Def. 6.2.2.8 und 9, zur Berechnung der Domänenerweiterung die Methode aus Def. 6.2.2.1 und 2 und zur erneuten Fixpunktberechnung die Methode aus Def. 6.2.2.5, die die Aufgaben jeweils delegieren.

Weitere Kommunikation zwischen Objekten (Variablen, Constraints und Solver) eines Constraintprogramms ist beim ACS (Kap. 3) nicht erforderlich.

□

6.3 Implementierung in J.CP

Zusammenfassung. Die derzeitige ACS Implementierung J.CP unterstützt die Implementierung verteilter Constraintprogramme durch Schnittstellen der Variablen- und Solver-Objekte für den *remote*-Zugriff. Die Schnittstellen werden durch Java RMI (*remote method invocation*) umgesetzt, das Methodenaufrufe in entfernten Prozessen durch *interfaces* ermöglicht. Das *interface* dient dabei jeweils als *proxy* für das eigentliche Objekt.

In der ACS Implementierung J.CP wird die Implementierung verteilter constraintbasierter Programme durch Erweiterungen der Klassen für Constraintvariablen und Solver ermöglicht. Die Kommunikation in der gegenwärtigen Implementierung nutzt das Konzept der *remote method invocation* (RMI) [Oeb01, RMI], das als Softwarepaket frei erhältlich ist [Java].

Die Grundidee von Java RMI liegt in der Abtrennung von Code, mit dem man Objekte benutzen kann von dem, der die Ojekte implementiert. Der Aufruf von Methoden und die Ausführung der Implementierung dieser Methoden kann in unterschiedlichen JVMs (*java virtual machine*), also in unterschiedlichen Prozessen stattfinden. Dazu werden *interfaces*, die das Verhalten für den *client* definieren und Klassen, die das Verhalten implementieren, verwendet. In diesem Zusammenhang sind unter dem Begriff *interface* abstrakte Klassen (oder *templates*) zu verstehen, bei denen in Java (im Unterschied zu normalen Klassen) auch Mehrfachvererbung möglich ist. Die Kommunikation zwischen *interface* und Objekt, also die Kommunikation zwischen entfernten Komponenten, wird dann intern durch RMI durchgeführt, indem das *interface* als *proxy* verwendet wird. Ein *proxy* ist ein *design pattern* aus [GHJV95], das ein Objekt eines Kontext in einem anderem Kontext repräsentiert. Das *proxy* weiß, wie die Methodenaufrufe zwischen den Objekten kommuniziert werden (vgl. Def. 6.2.2 und Seite 124). Das *interface* und die dazugehörige Klasse müssen dazu separat implementiert werden. Um dabei RMI zu nutzen, werden sie jeweils als Spezialisierungen von `java.rmi`-Klassen bzw. *interfaces* implementiert, wie es in Abb.6.7 für lokale bzw. *remote*-Variablen dargestellt ist. Der Aufruf von Methoden mit RMI ist standardmäßig synchron und wird daher immer dort genutzt, wo Synchonität notwendig ist. Zur Implementierung asynchroner Nachrichten mit RMI starte ich jeweils einen nebenläufigen *thread*, der der Methodenaufurf durchführt und sich anschließend beendet. Durch die Synchronisierung der RMI-Schnittstellen um Leser-Schreiber-Problemen vorzubeugen ergibt sich eine Warteschlage für asynchrone Nachrichten in jedem Agenten. Alternativ wäre auch eine Implentierung mit dem Java-Paket JMS (*java message service*) für asynchronen Nachrichtenaustausch über Kanäle möglich gewesen.

Aus dem *interface* und der entsprechenden Klasse werden mit dem `rmic`-Compiler die sogenannten *stub*- und *skeleton*-Klassen automatisch erzeugt. Objekte dieser Klassen vollziehen in verteilten Anwendungen die Kommunikation zwischen entfernten Komponenten über interne *sockets*. Diese benutzen einen eigenen Namensraum für mit RMI erreichbare Objekte, so daß solche Objekte jeder JVM mit dem TCP/IP Protokoll via Internet über einen Hardware-*port*

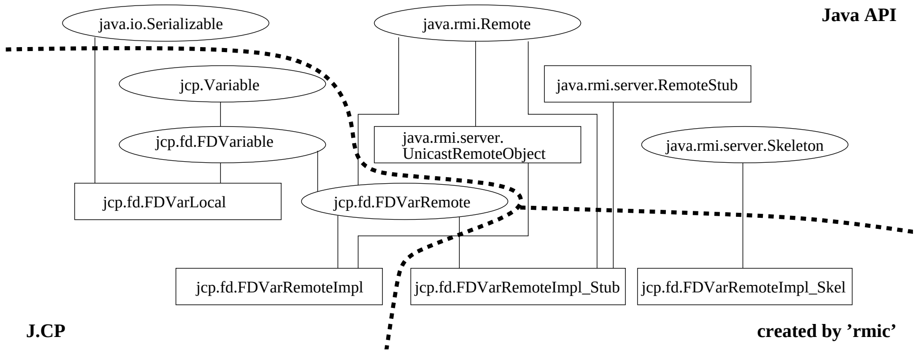

Abbildung 6.7: Integration von Variablen zur verteilten Implementierung in J.CP. Ovale beschreiben *interfaces* und Rechtecke Klassen. Jede FD-Variable kann auch durch *remote*-Zugriff über die Klasse FDVarRemoteImpl verwendet werden. In dieser Klasse werden alle Methodenaufrufe direkt an ein entsprechendes lokales FDVariable-Objekt delegiert (*delegation pattern*).

erreichbar sind. Die Adresse und der *port* müssen dem *client* vor der eigentlichen Kommunikation mitgeteilt werden, danach kann auf die entfernten Objekte genauso zugegriffen werden, wie auf lokale [RS02].

Fallstudie, Teil 22

Bevor in $\mathcal{M}$Plan ein gemeinsamer Termin festgelegt werden kann, muß der anfragende Agent die Adresse und den port des gewünschten Terminpartners kennen. Die Adresse (z.B. `rmi://rhein.first.fhg.de/gr`*) setzt sich zusammen aus dem Namen des Computers (*`rhein.first.fhg.de`*) und dem des Benutzers (*`gr`*), so daß jeder Benutzer auf jedem Computer genau einen Agenten benutzen kann. Als Hardware-port wird der für Java reservierte port 1099 angenommen, wenn kein anderer explizit angegeben wurde (eine Adresse mit explizitem port wäre z.B.* `rmi://rhein.first.fhg.de:1099/gr`*). Mit einer Anfrage wird dann die eigene Adresse übermittelt, so daß das Appointment-Objekt app das mit dem Methodenaufruf von howAbout(app) (Abb. 6.6) übermittelt wurde für den remote-Zugriff zur Verfügung steht. Damit können alle Methoden, die in Teil 19 der Fallstudie in den interfaces dargestellt wurden, von entfernten Objekten aufgerufen werden.*

6.4 Verteilte Suche in ACS

Zusammenfassung. Die vorgestellten Suchalgorithmen eignen sich nur bedingt zur verteilten Anwendung, weil sie einige zentrale Steuerung erfordern. Es ist zu erwarten, daß chronologisches *backtracking* und die vorgestellte *local search* durch den hohen Kommunikationsaufwand und weil keine parallelen Berechnungen durchgeführt werden, ineffizienter als echte verteilte Suchalgorithmen sind. Bei den bekannten Algorithmen zur verteilten Suche ist lediglich eine zentrale Instanz nötig, die stabile globale Zustände, die z.B. durch Finden einer Lösung erreicht werden, erkennt. Die Integration solcher Algorithmen in ACS wird erst im Ausblick dieser Arbeit beschrieben. In diesem Abschnitt werden lediglich die durch Verteilung für die Suche entstehenden Probleme beleuchtet und die Lösung für $\mathcal{M}$Plan durch unvollständige Suche angegeben.

Die Suchconstraints aus Abschnitt 5.4 sind nur bedingt zum Lösen verteilter CSPs geeignet. Die Algorithmen können zwar, wie jedes andere Constraint des J.CP auch, verteilt verwendet werden, dabei wird aber mehr Kommunikation über Prozessgrenzen hinweg durchgeführt als nötig. Bei der Verwendung für verteilte Probleme werden die Instantiierungen und das Suchconstraint in einem Solver abgesetzt und benutzen die *remote*-Schnittstellen der Variablen, um Domänen entfernter Variablen einzuschränken.

Beim chronologischen *backtracking* (Abschnitt 5.4.1) und beim *local search* (Abschnitt 5.4.2) sind in verteilten Anwendungen Nachrichten zur Domänendefinition und für Domänenanfragen erforderlich. Die entsprechenden Methoden stehen zwar laut Def. 6.2.1.1 und 2. zum *remote*-Zugriff zur Verfügung, aber es ist bei komplexen verteilten Problemen ineffizient, alle Information bei der Suche zentral zu verwalten. Verteilte Suchalgorithmen hingegen nutzen die Möglichkeit, die Effizienz durch parallele Berechnungen zu steigern. Obwohl alle Hilfsconstraints des *labeling* in einem Solver abgesetzt werden, müssen die Inkonsistenzen über das physische Netzwerk kommuniziert werden (Def. 6.2.1.6). Denn durch Propagation kann es in jedem Agenten zu Inkonsistenzen kommen, die durch eine bessere Wertauswahl im *labeling* beseitigt werden können und sollen. Weil im Algorithmus sichergestellt sein muß, daß die Propagation abgeschlossen ist bevor die nächste Variable instantiiert wird (Abschnitt 5.3.1), ist außerdem das Prüfen eines verteilten Zuverlässigkeitswerts notwendig (Def. 6.2.1.3). Erst wenn alle erreichbaren Solver volle Zuverlässigkeit erreicht haben, ist die Propagation abgeschlossen und die Suche kann fortgesetzt werden. Desweiteren kann es durch die Kommunikation von Variablendomänen zur Verletzung von *privacy*-Anforderungen kommen, weil die Variablendomänen beim *client* damit sichtbar sind. Für die Anwendung $\mathcal{M}$Plan würde die Verwendung der zentralen Suchalgorithmen erlauben, daß ein Benutzer den Terminkalender anderer Benutzer durch mehrfache Anfragen rekonstruieren kann. Bei sicherheitsrelevanter Software müssen außerdem noch die übermittelten Daten verschlüsselt werden, was die Kommunikation nochmals verlangsamt.

Um DCSPs lösen zu können, die aufgrund dieses hohen Kommunikationsaufwands oder *privacy/security*-Anforderungen nicht durch die zentralen Suchalgorithmen aus Abschnitt 5.4 gelöst werden können, ist die Implementierung verteilter Algorithmen nötig. Die bekannten Suchalgorithmen [YD98, YH00] starten in den Prozessen der zu instantiierenden Variablen jeweils Agenten, die die Instantiierung und Konsistenzprüfung lokal durchführen. Dazu muß das ACS Ausführungsmodell so modifiziert werden, daß Hilfsconstraints in frei wählbaren Solvern abgesetzt werden können. Durch eine solche Anpassung von Def. 3.3.6.2 muß auch die Berechnung des Zuverlässigkeitswertes (Def. 3.3.4.7) angepaßt werden.

Definition 6.4.1 (Verteilte Zuverlässigkeit) In einem Verteilten Constraintprogramm P berechnet sich die Zuverlässigkeit eines Solver s (also den Wert $x = s.\texttt{getReliability}())$ als Minumum der Zuverlässigkeit aller im Constraintnetz erreichbaren Solver durch $x := min(s'.\texttt{getReliability}())$ mit $s' \in \{\mathsf{solvers}(P) \mid \exists\{c_1, ..c_n\} \subseteq \mathsf{constraints}(P) : (c_1.solver = s \wedge c_n.solver = s' \wedge \forall 2 \leq i \leq n : \exists v \in \mathsf{vars}(c_i) \cap \mathsf{vars}(c_{i-1}))$

Durch diesen verteilten Zuverlässigkeitswert kann auch in verteilten ACS Programmen die Existenz eines Ergebnisses (Def. 3.4.3) überprüft werden. Dadurch ist auch die Termination von Propagation und Suche feststellbar, die einen stabilen Zustand (gescheitert, arc-konsistent oder Lösung gefunden) sichert. Die verteilte Zuverlässigkeit läßt sich zentral prüfen, wenn alle Agenten in einer zentralen Anwendung bekannt sind. Unter der Annahme, daß jede Nachricht irgendwann ankommt, kann gezeigt werden, daß ein verteiltes Constraintprogramm genau dann terminiert, wenn seine Zentralisierung terminieren würde.

Lemma 6.4.1 Wenn die Zentralisierung Z eines Verteilten Constraintprogramms P ein Ergebnis hat, dann hat auch P ein Ergebnis.

Beweis:
Wenn die Zentralisierung Z ein Ergebnis hat, so gilt $\forall(z, A, \rho) \in \mathsf{solvers}(Z) : z = \top$, dies bedeutet aber laut Def. 6.4.1, daß für jeden Solver $(z'.A', \rho') \in \mathsf{solvers}(P)$ ebenfalls $z' = \top$ gilt. Damit sind keine keine weiteren Berechnungen mehr möglich und P hat ein Ergebnis.

□

Zum Finden stabiler globaler Zustände von einer zentralen Instanz aus, wurden im Bereich der verteilten Programmierung effiziente und sichere Algorithmen, wie z.B. *distributed snapshots* [CL85, Mat93] entwickelt. Bei der Implementierung verteilter Suchalgorithmen wie *asynchronous backtracking* oder dem verteilten *local search* Algorithmus *distributed breakout* [YH00] wird eine solche zentrale Kontrolle in dem Agenten gestartet, in dem die Suche gestartet wurde. Für eine ACS Implementierung der genannten Algorithmen bedeutet das, daß die Zuverlässigkeit aller an der Suche beteiligten Agenten zentral geprüft wird, um die Terminierung des Algorithmus festzustellen. Erst wenn der verteilte Algorithmus terminiert, ist eine Lösung in den aktuellen Variablendomänen ablesbar.

In offenen Constraintnetzen, wie sie in Agenten-basierten Systemen, wie z.B. der verteilten Terminplanung entstehen, kann die verteilte Zuverlässigkeit nicht von einer zentralen Stelle kontrolliert werden, weil durch die unbekannte Vernetzung i.d.R. nicht alle Agenten bekannt sind. In solchen Anwendungen müßten die betroffenen Solver durch Traversierung des Constraintnetzes erst gesucht werden. Alternativ könnte, wie es in [Sch02] beschrieben wird, eine lokale Kopie des Suchraums erzeugt werden, in dem die Suche durchgeführt wird. Das

Ergebnis dieser gekapselten Suche könnte dann in Form von Instantiierungen abgesetzt werden. Dazu wäre ein Konsistenztest in den Constraints nötig, der keine Variableneinschränkungen vornimmt, so daß die Konsistenz während der Suche geprüft wird, ohne Inkonsistenzen im Sinne von Def. 3.3.5.4 zu erzeugen bzw. zu verarbeiten. Im Ausblick dieser Arbeit (Abschnitt 8.2) werden diese Ideen nochmals aufgegriffen.

Fallstudie, Teil 23
In $\mathcal{M}$Plan ist kein vollständiger Suchalgorithmus durch die Reparaturoperationen der einzelnen Agenten gegeben. Stattdessen wird auf Inkonsistenzen lokal so reagiert, daß unwichtige Termine verschoben oder abgesagt werden. Das Constraint `LabelBefore(Appointment a, Calendar d)` *setzt einen Termin* `a` *in Beziehung zu einem bestimmten Zeitpunkt* `d`*, an dem der Termin festgelegt werden soll. In Java wird mit Objekten der Klasse* `Calendar` *ein Zeitpunkt definiert und nicht etwa ein Kalender. In regelmäßigen Abständen prüft dann in jedem Agent ein dafür vorgesehener thread, ob das Instantiierungsdatum eines Termins überschritten ist. Ist dies der Fall, so wird der Termin durch das in Teil 12 beschriebene Constraint instantiiert. Dazu wird ein verteiltes Suchconstraint abgesetzt (Teil 13 der Fallstudie), das für eine bestimmte Menge erreichbarer Solver versucht, die beste Instantiierung zu finden (Teil 14). Die Instantiierung kann aber durch Propagation zu Inkonsistenzen, also zu Verletzungen von* `Disjoint-` *Constraints in weiteren, im Constraint unberücksichtigten Agenten führen, auf die standardmäßig durch folgende Reparaturoperation lokal reagiert wird:*

```
class StandardListener extends InconsistencyListener{
   void handleInconsitency(InconsistencyEvent e){
      // das gescheiterte Constraint, ist in e gespeichert
      Disjoint failed = e.getConstraint();
      Appointment a1,a2;            // die kollidierenden Termine
      a1 = failed.vars[0];  a2 = failed.vars[1];
      int p1,p2;  // Die Wichtigkeit/Anzahl der Teilnehmer
      p1 = evalPriority(a1.attendees);
      p2 = evalPriority(a2.attendees);
      Instantiation inst; // inst soll anders gewaehlt werden
      if(p1 > p2)  // zu a1 kommen wichtigere Leute
         inst = a2.getInstantiation();
      else // zu a2 kommen wichtigere Leute
         inst = a1.getInstantiation();
      if(inst != null)
          inst.retract(); // Instantiierung loeschen...
            // ...und neu suchen
          (new Suche(inst.vars[0])).post(mySolver);
      else // es gab eine Inkonsistenz, obwohl noch nicht
          // instantiiert d.h. der Kalender fuer diesen Tag ist voll
          // und es muss ein unwichtiger Termin verschoben werden
          Instantiation other = a1.getDay.getUnimportantMeeting();
          other.retract();
          (new Suche(other.vars[0])).post(mySolver);
   }
}
```

Es wird also versucht, einen unwichtigen Termin umzuplanen, indem seine Instantiierung zurückgenommen wird und erneut ein Suchconstraint ausgeführt wird. Hierbei sollen endlose Zyklen von Umplanungen abhängiger Termine dadurch verhindert werden, daß der Kontext des ausführenden Agenten i.d.R. ein anderer ist, als der des Constraints, das die Inkonsistenz verursacht hat. Wird der Kontext, in dem die Inkonsistenzen schon während der Suche abgefangen werden, groß genug gewählt (Fallstudie 14), so sind kaum Zyklen durch immer wiederkehrende Suche und Constraintrücknahme zu erwarten, wenngleich das theoretisch nicht ausgeschlossen werden kann.

Kapitel 7

Vergleiche und Benchmarktests

In diesem Kapitel wird das ACS Ausführungsmodell anhand der Eigenschaften seiner prototypischen Implementierung J.CP evaluiert. Neben einigen Laufzeitvergleichen der unterschiedlichen Suchconstraints sowie mit anderen Systemen, wird vor allem der Effekt der asynchronen Ausführung untersucht. Es hat sich herausgestellt, daß die Reihenfolge, in der die Propagationsmethoden abgesetzter Constraints im Solver ausgeführt werden, großen Einfluß auf die Performanz des Systems hat. Bei der Bearbeitung klassischer *benchmark*-Probleme ist J.CP der effizienteste, frei erhältliche Solver und ist nur wenig langsamer als das professionelle SICStus CLP-System.

In der prototypischen ACS Implementierung J.CP stehen einige einfache Constraints über endliche Mengen ganzer Zahlen (`jcp.fd`) und über Mengen beliebiger Objekte (`jcp.enum`) zur Verfügung. Mit diesen Constraints lassen sich einige klassische Constraintprobleme, wie z.B. viele Benchmarkprogramme aus [GW] lösen. Damit können aber lediglich die Laufzeiteigenschaften eines Solvers bei fest gegebenen CSPs in monolithischen Anwendungen eines Constraintsystems (meistens *finite domain*) untersucht werden. Die Performanz bei solchen Problemen ist aber aus softwaretechnischer Sicht nur eine Anforderung an Constraintlöser (siehe Abschnitt 3.1), so daß ich erst am Ende des Kapitels darauf eingehen werde. Die anderen Anforderungen, also die Möglichkeiten, die der Solver bzw. dessen Ausführungsmodell bei der Implementierung von Constraintprogrammen zur Verfügung stellt, wurden im Verlauf der Arbeit immer wieder erwähnt und in Abschnitt 7.3 zur Evaluation einiger bekannter Systeme verwendet.

Da J.CP zunächst zur Evaluierung des Ausführungsmodells benutzt werden soll, wurde bei der Implementierung viel Wert auf generische Komponenten, wie z.B. unterschiedliche Aufgabenpuffer, gelegt. Der damit teilweise verbundene Performanzverlust wird in der aktuellen Version in Kauf genommen, um die Auswirkungen der unterschiedlichen Komponenten auf die Laufzeit in Anwendungsbeispielen zu untersuchen. In den folgenden beiden Abschnitten werden daher zunächst Vergleiche unterschiedlicher Implementierungen von J.CP angestellt, um die Vor- und Nachteile des Ausführungsmodells zu evaluieren.

Alle Laufzeituntersuchungen wurden auf einem Dual Pentium III-500 Linux PC durchgeführt. Der Solver wurde mit der Java Implementierung jdk 1.3 von sun microsystems [Java] ausgeführt. Die anderen untersuchten Solver wurden jeweils in der neuesten erhältlichen Version bzw. ebenfalls mit jdk 1.3 verwendet, die Versionsnummern werden jeweils angegeben.

Da durch die asynchrone Ausführung und insbesondere auch bei verteilten Anwendungen und den vorhandenen Suchalgorithmen die Programmausführung nichtdeterministisch ist, werden alle Testwerte als Mittelwert von mindestens drei Testläufen angegeben. Da jedes ACS Programm mindestens zwei getrennte *threads* verwendet, nämlich den des Solvers und den der Anwendung, sind die Laufzeiten normalerweise unterschiedlich. Allerdings waren bei den Anwendungen ohne Suche keine signifikanten Unterschiede zu erkennen, so daß die Varianz dort nicht angegeben wird. Bei den Suchalgorithmen, insbesondere bei *local search* waren größere Unterschiede zu messen, was in Abschnitt 7.2 noch genauer beschrieben wird.

7.1 Propagation

Zusammenfassung. In diesem Abschnitt wird der Einfluß der asynchronen Ausführung von Arbeitsaufgaben aus dem Aufgabenpuffer der Constraintlöser untersucht. Dazu wird die Laufzeit der Propagation (ohne Suche) in unterschiedliche Arten von Puffern anhand eines *benchmark*-Programm verglichen. Der Einfluß den die Priorisierung von Aufgaben auf die Laufzeit der Propagation hat, wird ebenfalls untersucht. Schließlich wird dargestellt, wie sich die Laufzeiten mit einem zentralen Solver, mit nebenläufigen *threads* und mit nebenläufigen Prozessen verändern.

Beim ACS spielt die Propagation von Domäneneinschränkungen eine wesentliche Rolle, weil durch sie Lösungen gefunden bzw. Inkonsistenzen entdeckt werden. Die Suchalgorithmen sind lediglich dazu da, die Instantiierungen zu finden, deren Propagation zu keinen Inkonsistenzen führt. Die Propagation, sowohl *look-ahead* als auch *forward-checking*, wird beim Constraintlösen wesentlich von der Reihenfolge, in der die Domäneneinschränkungen vorgenommen werden, beeinflußt. Beim ACS braucht diese Reihenfolge nicht durch die Programmiererin entsprechend ihrer Erfahrung festgelegt zu werden, sondern wird durch den verwendeten Aufgabenpuffer bestimmt (Def. 3.2.2). Dieser kann bei J.CP individuell für jede Anwendung definiert werden, was die Laufzeit bei bestimmten Problemen erheblich verringern kann (vgl. optimaler Puffer in Abb. 7.3). In der aktuellen Version stehen 4 verschiedene Puffertypen zur Verfügung, die im folgenden untersucht werden. Außerdem kann jeder Solver auch synchron benutzt werden. Wird ein entsprechender Schalter aktiviert, so werden die Propagationsmethoden der abgesetzten Constraints immer sofort ausgeführt.

Unterschiedliche Aufgabenpuffer

Um die unterschiedlichen Puffertypen zu vergleichen, habe ich ein *benchmark* Programm verwendet, dessen Constraints alle die gleiche Priorität haben, so daß kein Expertenwissen über den potentiellen Effekt der Constraints Einfluß auf die Ergebnisse haben kann.

Das untersuchte *order*-Problem besteht aus n Variablen $\{v_1, ..., v_n\}$, die jeweils die initiale Domäne ganzer Zahlen $\{1, 2, ..., n\}$ haben. Dann werden n Constraints abgesetzt, die jeweils zwei dieser Variablen in eine "kleiner-als" Beziehung setzten: $\forall 1 \leq i \leq n-1 : v_i < v_{i+1}$. Die einzige Lösung für dieses Problem ist $\forall 1 \leq i \leq n : v_i = i$, die durch *look-ahead* Propagation und ohne Suche gefunden werden kann. Jedes $x < y$-Constraint setzt die obere Grenze der Domäne von x auf $max(y) - 1$ und die untere Grenze von y auf $min(x) + 1$. Das Constraint wird erneut ausgeführt, wenn sich die obere Schranke von y oder die untere Schranke von x ändert.

In J.CP stehen folgende Klassen als Aufgabenpuffer zur Verfügung, die jedem Solver bei jeder Anwendung individuell zugeordnet werden können:

1. Stack: Eine herkömmliche LIFO Datenstruktur, die zuletzt abgelegten Daten werden zuerst weiterverarbeitet. Die Ein- und Ausgabeoperationen bei diesem Puffer sind sehr effizient.

2. HashSet: Die Daten werden in einer Hashtabelle abgelegt, wobei jede Aufgabe nur genau einmal eingefügt wird, d.h. das Einfügen eines Datensatzes, der bereits enthalten ist, führt zu keinen Veränderungen. In welcher Reihenfolge die Daten ausgegeben werden, ist unbestimmt. Die Ein- und Ausgabeoperationen bei diesem Puffertyp sind etwas aufwendiger, weil bei der Eingabe geprüft wird, ob der Datensatz bereits im Puffer enthalten ist und bei der Ausgabe, ob noch einer zur Ausgabe vorhanden ist.

3. StackSet: Wie Stack, mit der Erweiterung, daß Datensätze nur gespeichert werden, wenn sie nicht bereits enthalten sind. Die Eingabe von Daten ist daher ineffizienter als bei Stack.

4. QueueSet: Eine FIFO Warteschlange, bei der der Datensatz zuerst ausgegeben wird, der zuerst eingegeben wurde. Datensätze, die bereits enthalten waren, werden nicht nochmals eingefügt. Die Ein- und Ausgabe ist genauso effizient wie bei StackSet.

5. Synchronous: Die Aufgaben werden nicht im Puffer abgelegt, sondern wie beim klassischen Constraintlösen (z.B. CLP) direkt ausgeführt. Hierfür ist ein entsprecheder Schalter in der Solver-Klasse von J.CP vorgesehen. Da kein Puffer verwendet wird, entsteht durch ihn auch kein Verwaltungsaufwand, die Propagationsmethoden werden direkt vom Solver aufgerufen.

Die unterschiedliche Laufzeit des *order-benchmark* ist in Abb. 7.1 in Abhängigkeit der Eingabegröße n dargestellt.

Durch Variationen der Ausführungsreihenfolge der Propagationsmethoden der Constraints, kann es (auch) bei diesem Problem zu Unterschieden bei der Propagation kommen. Im ungünstigsten Fall werden die Domänen der Variablen bei jeder Ausführung eines Constraints nur um genau einen Wert eingeschränkt, so daß bei der Ausführung und bei der erneuten Ausführung eines Constraints alle anderen Constraints ebenfalls erneut ausgeführt werden. Wenn bei einer Ausführung gleich mehrere Werte aus der Domäne entfernt werden können, brauchen nur manche Constraints daraufhin erneut ausgeführt werden. Diese Effekte bringen mit wachsender Problemgröße mehr Laufzeitgewinn, aber auch

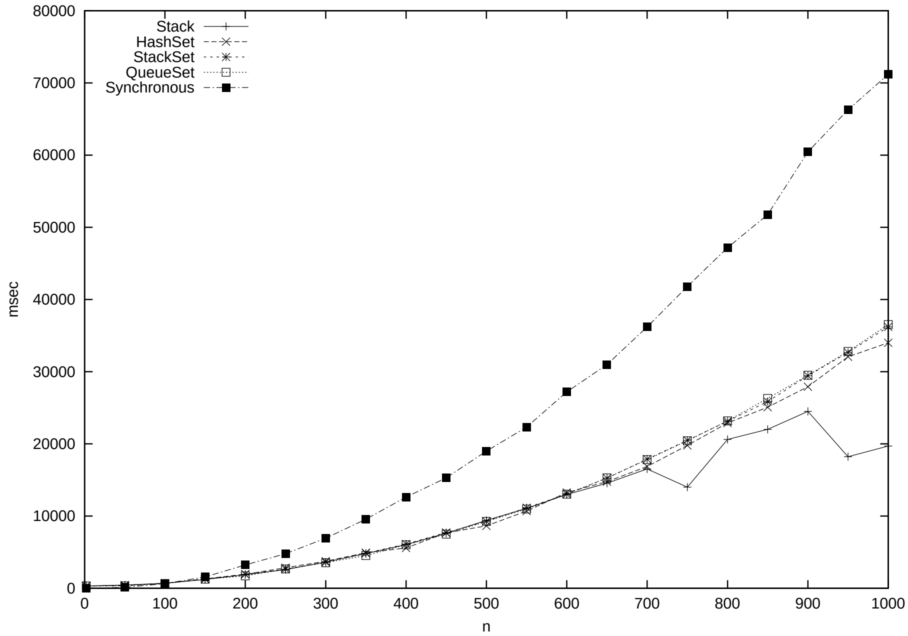

Abbildung 7.1: Die Laufzeiten zur Lösung des *order* Problems mit unterschiedlichen Aufgabenpuffern

schon bei $n = 4$ kann die Propagation um die Ausführung eines Constraints durch eine geschickte Ausführungsreihenfolge der Propagationsmethoden reduziert werden, wie es in Abb. 7.2 dargestellt wird.

Bei den verwendeten Aufgabenpuffern in Abb. 7.1 ist zu beobachten, daß Puffer, die erneut abgesetzte Constraints später ausführen als solche, die zum erstenmal ausgeführt werden (z.B. bei Stack), das Problem schneller lösen, als solche, die die Propagation sofort ausführen (z.B. bei QueueSet). Dies ist durch die beschriebenen "Synergieeffekte" (Abb. 7.2) zu erklären. Desweiteren ist die synchrone Ausführung (Synchronous) langsamer als die mit Warteschlange, obwohl beide die ungünstigste Reihenfolge benutzen. Dies ist durch die Implementierung des Puffers als Menge zu erklären, jede Propagationsmethode wird nur einmal gepuffert. Dadurch werden manche Aufgaben, die wegen eines Constraints erneut ausgeführt werden müssen und noch nicht zur Ausführung kamen, bei der Propagation eines anderen Constraints nicht erneut gepuffert.

Die Prioritäten der Aufgaben

Durch die Definition der Wichtigkeit jeder Propagationsmethode eines ACS Constraints (Def. 3.2.1) kann die Ausführungsreihenfolge der Methoden ebenfalls beeinflußt werden. Je höher die Priorität einer Aufgabe im Puffer (Def. 3.2.2) ist, desto früher soll sie ausgeführt werden.

In J.CP können solche Prioritäten für jede Propagationsmethode eines Constraints bzw. für seine Rücknahmeoperation individuell angegeben werden. Die implementierten Aufgabenpuffer, die auf Seite 139 vorgestellt wurden, verwenden diese Prioritäten, indem Aufgaben einer Priorität immer erst dann aus-

normal	optimal
$x_1 \in 1..4, x_2 \in 1..4, x_3 \in 1..4, x_4 \in 1..4$	$x_1 \in 1..4, x_2 \in 1..4, x_3 \in 1..4, x_4 \in 1..4$
$x_1 < x_2$ (from buffer) $x_1 \in 1..3, x_2 \in 2..4, x_3 \in 1..4, x_4 \in 1..4$	$x_1 < x_2$ (from buffer) $x_1 \in 1..3, x_2 \in 2..4, x_3 \in 1..4, x_4 \in 1..4$
$x_2 < x_3$ (from buffer) $x_1 \in 1..3, x_2 \in 2..3, x_3 \in 3..4, x_4 \in 1..4$	$x_2 < x_3$ (from buffer) $x_1 \in 1..3, x_2 \in 2..3, x_3 \in 3..4, x_4 \in 1..4$
$x_1 < x_2$ (propagation) $x_1 \in 1..2, x_2 \in 2..3, x_3 \in 3..4, x_4 \in 1..4$	$x_3 < x_4$ (from buffer) $x_1 \in 1..3, x_2 \in 2..3, x_3 = 3, x_4 = 4$
$x_3 < x_4$ (from buffer) $x_1 \in 1..2, x_2 \in 2..3, x_3 = 3, x_4 = 4$	$x_2 < x_3$ (propagation) $x_1 \in 1..3, x_2 = 2, x_3 = 3, x_4 = 4$
$x_2 < x_3$ (propagation) $x_1 \in 1..2, x_2 = 2, x_3 = 3, x_4 = 4$	$x_1 < x_2$ (propagation) $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4$
$x_1 < x_2$ (propagation) $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4$	

Abbildung 7.2: Vergleich der optimalen und der normalen Ausführungsreihenfolge der Propagationsmethoden beim *order 4* Problem. Bei der normalen Reihenfolge wird die Propagation immer sofort ausgeführt, was bei diesem Problem ungünstiger ist, als zunächst weiteres Wissen (andere Constraints) zu verarbeiten.

geführt werden, wenn alle abgesetzten Aufgaben mit höherer Priorität abgeschlossen sind. In zukünftigen Versionen von J.CP soll zusätzlich die Reihenfolge, in der Aufgaben abgesetzt wurden bei der Auswahl der nächsten Aufgabe berücksichtigt werden, im eventuelles Aushungern [HH89] von Aufgaben mit niedriger Priorität zu verhindern.

Mit der aktuellen Implementierung kann aber durch eine entsprechende Definition der Prioritäten in den Constraints, das Erreichen einer bestimmten Art der Konsistenz (vgl. Def. 2.2.2) durch den Zuverlässigkeitswert des Solvers angezeigt werden. Wenn z.B. die Priorität aller Propagationsmethoden aller unären Constraints den Wert x hat, so kann das Erreichen des Zuverlässigkeitswertes $max - x$ in der Solvern als Knotenkonsistenz interpretiert werden, denn bei diesem Zuverlässigkeitswert wurden alle relevanten Constraints ausgeführt und propagiert.

Um den Einfluß der Prioritäten auf die Performanz bei der Propagation zu testen, habe ich ebenfalls ein *benchmark*-Programm verwendet, das keine Suche und keine komplexen Propagationsalgorithmen verwendet. Das *reorder-n*-Problem ist gegeben durch ein *order*-Problem, bei dem nach dem Absetzen aller $<$-Constraints, das n-te Constraint zurückgenommen wird. Bei m Variablen enthält die Domäne jeder Variable bei abgeschlossener Propagation dann entweder $m - n$ oder $n - m$ Elemente. Das Problem ist also mit weniger Propagation zu lösen als das *order*-Problem, wenn die Reihenfolge der Aufgaben entsprechend angepaßt wird. In Abb. 7.3 und Abb. 7.4 sind die Laufzeitergebnisse in Millisekunden in Abhängigkeit von der Anzahl der Variablen angegeben. Dabei wird in 7.3 das $n/2$-te, im ersten Diagramm von 7.4 das erste und im zweiten Diagramm dort das n-te $<$-Constraint zurückgenommen. Zusätzlich zu den bekannten Aufgabenpuffern wurde ein optimaler Puffer für dieses Problem verwendet, der am Ende dieses Abschnitts vorgestellt wird.

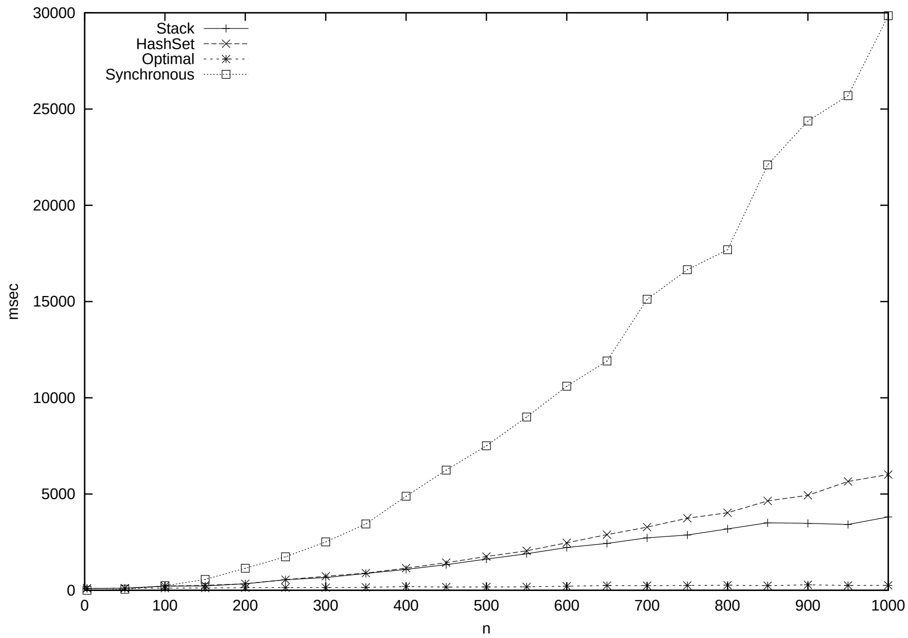

Abbildung 7.3: Die Laufzeiten zur Lösung des *reorder-n/2* Problems mit unterschiedlichen Aufgabenpuffern und Prioritäten

In der Implementierung von *reorder* werden die Prioritäten von $<$-Constraints kleiner angesetzt als die der Constraintrücknahme. Diese Festlegung ergibt sich aus Expertenwissen über das Problem und über Propagation: "Veraltetes Wissen (das zurückzunehmende Constraint) soll so früh wie möglich aus dem System (Problem) entfernt (zurückgenommen) werden". Dadurch kann die Rücknahmemethode, obwohl sie als letztes abgesetzt wird, ausgeführt werden, bevor alle Constraints fertig propagiert haben. Bei der Lösung des *reorder-2* Problems ist diese Priorisierung besonders effektiv, weil die Rücknahme eines so früh abgesetzten Constraints sonst aufwendiger wäre als z.B. die Rücknahme des letzten Constraints, wie bei *reorder-n*. In den Diagrammen ist aber zu erkennen, daß die Ausführung beider Probleme in etwa gleich lang dauerte. In Vergleichstests mit dem CLP-Solver SICStus war die Rücknahme[1] des ersten Constraints wesentlich aufwendiger als die des letzten, weil dort das gesamte Problem ein zweites mal, ohne das erste Constraint, berechnet wird.

Das *reorder-n/2* (Abb. 7.4) Problem kann schneller als die anderen *reorder* Varianten und auch schneller als das *order*-Problem gelöst werden, weil insgesamt nur die Hälfte der Propagation notwendig ist. Bei diesem Problem enthält die Domäne jeder der m Variablen genau $m/2$ Werte, es brauchten also nur halb so viele Werte durch Propagation entfernt werden wie bei *order*.

[1] Constraintrücknahme kann in zustandsbasierten Systemen, wie z.B. CLP, durch Wiederherstellung des Zustands vor dem Absetzten des zurückzunehmenden Constraints und durch wiederholtes Absetzen aller darauf folgenden Constraints umgesetzt werden.

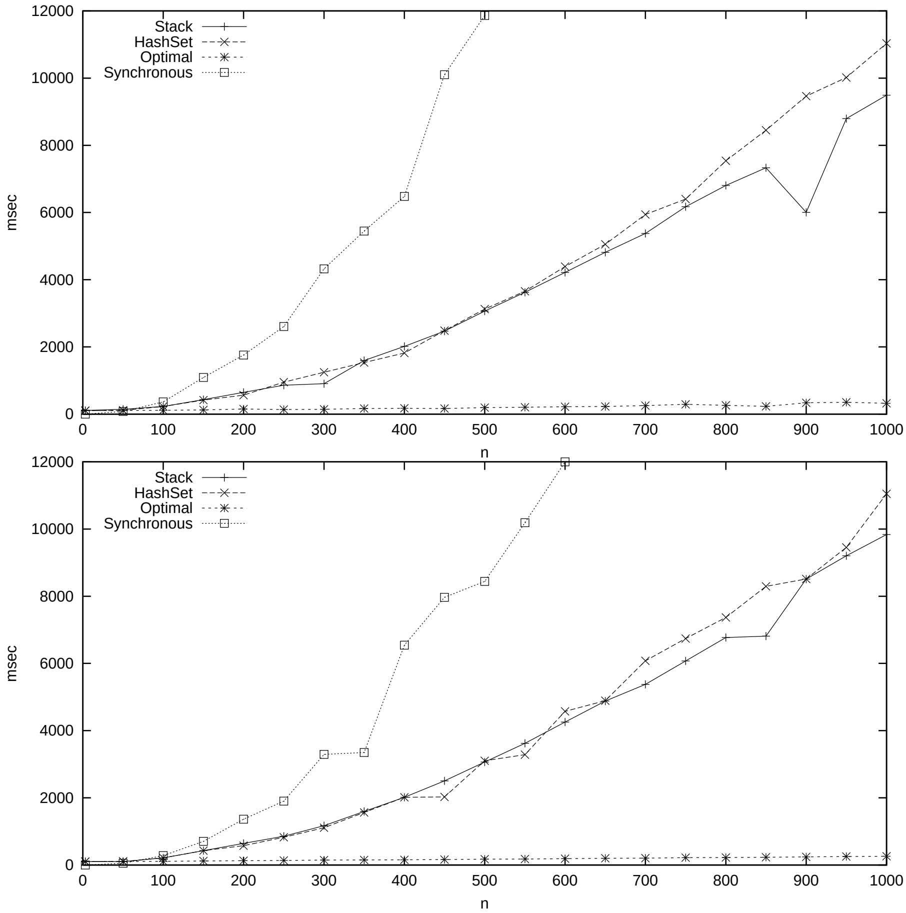

Abbildung 7.4: Die Laufzeiten zur Lösung des *reorder-2* und des *reorder-n* Problems mit unterschiedlichen Aufgabenpuffern und Prioritäten

Durch die einfache Problemstruktur kann die Ausführung bei den *order*- und *reorder* Problemen mit Expertenwissen extrem verbessert werden. Dies wurde durch die Verwendung zweier Aufgabenpuffer für die <-Constraints in der mit "Optimal" gekennzeichneten Kurve in Abb. 7.3 und Abb. 7.4 dargestellt. Bei dieser Programmausführung wurden die <-Constraints durch zwei Propagationsmethoden mit unterschiedlicher Priorität implementiert. Die eine Methode eines $x < y$-Constraints behandelt den maximalen Wert von x, die andere den minimalen Wert von y. Für die Aufgaben, die die beiden Methoden referenzieren, wurde ein separater Puffer verwendet, indem beiden Aufgaben unterschiedliche Prioritäten zugewiesen wurden. Dadurch konnten zuerst alle unteren Grenzen und erst danach alle oberen Grenzen definiert werden. Durch die Verwendung einer Warteschlange für die Propagation der unteren Grenzen und eines Stack für die der oberen, braucht jedes Constraint nur genau zweimal ausgeführt werden ($O(2n)$), was eine erhebliche Verbesserung gegenüber der normalen Propagation darstellt. Bei der normalen Propagation, mit den anderen Puffertypen, muß bei jedem Constraint die untere Grenze bis zur letzten Variable und die obere Grenze bis zur ersten Variable propagiert werden, so daß sich eine Komplexität von $O(n^2)$ ergibt. Durch ähnliche Überlegungen und die Möglichkeit, die Puffer und Prioritäten individuell festzulegen, können evtl. auch in praktischen Anwendungen solche Komplexitätsverbesserungen erreicht werden.

Propagation in verteilten Anwendungen

Das *order* Problem kann auf mehrere Agenten verteilt werden, indem jedem Agent ein Teil der Variablen zugewiesen wird, deren Domänen von lokalen Constraints reduziert werden. Zwischen jedem Paar von Agenten wird genau ein Constraint abgesetzt, so daß das Problem insgesamt gelöst wird. Die Agenten können entweder als nebenläufige *threads* oder als Prozesse umgesetzt werden. Ich habe beides untersucht, wobei auch bei der Implementierung mit *threads* keine Zugriffe auf gemeinsamen Speicher durchgeführt wurden.

In Abb. 7.5 und Abb. 7.6 habe ich die Ergebnisse der Laufzeituntersuchungen mit 1,2,5 und 10 *threads* dargestellt. Es wurden jeweils $\frac{n}{|threads|}$ Variablen in jedem Agent verwaltet. Die jeweils letzte und erste Variable wurde als Java RMI-Objekt definiert, um die verteilte Kommunikation zwischen den Agenten durch ein <-Constraint über diesen Variablen durchzuführen. Dadurch, daß Constraints über lokalen Variablen in J.CP mit höherer Priorität verarbeitet werden als solche über *remote*-Variablen, werden bei der Ausführung meist zuerst die lokalen (Teil-)Probleme gelöst und erst danach die Einschränkungen, die sich aus den Berechnungen entfernter Agenten ergeben, verarbeitet. Hierdurch ergibt sich eine implizite Trennung von innerem und äußerem Constraintlösen, die bei [Han01] explizit vorgenommen werden muß, aber bei ACS keine zusätzliche Koordination erfordert.

Die Ausführungszeit im Test aus Abb. 7.5 mit mehreren nebenläufigen *threads* ist bei großen n wesentlich besser als die monolithische. Nur bei kleinen n ist die Verwaltung der *threads* und die aufwendigere Kommunikation mittels RMI in der Laufzeit negativ bemerkbar. Da die *threads* alle auf einer 2-Prozessor Maschine ausgeführt wurden, kann der Unterschied bei großen n nicht an der parallelen Berechnung liegen und ist daher durch die Problemstruktur zu erklären. Wenn alle Constraints in einem Solver ausgeführt werden, so können bei der Ausführung der Propagationsmethoden jeweils nur kleine Domäneneinschränkungen vorgenommen werden. Wenn das Problem hingegeben z.B. bei $n = 1000$ und 10 Agenten gelöst wird, so können zuerst in jedem Agent die Domänen von jeweils 100 Variablen durch "kleine" Einschränkungen reduziert werden, bevor eine "große" Reduktion um 99 zum benachbarten Agenten kommuniziert wird. Diese große Reduktion führt, wie es im Abschnitt über die Abhängigkeit von Performanz und Aufgabenpuffer diskutiert wurde, zu erheblichen Laufzeitverbesserungen. Hier zeigt sich also, daß die Definition einer günstigen Konfiguration [Han01] die Laufzeit von nebenläufigen Constraintprogrammen erheblich verbessern kann.

Um die Performanz bei der nebenläufigen Lösung von *order* unabhängig von diesen Effekten zu untersuchen, habe ich den gleichen Test unter Verwendung des optimalen Aufgabenpuffers aus dem vorherigen Abschnitt durchgeführt. Durch die Verwendung dieses Puffers ist die Ausführungsreihenfolge in jedem Agenten bereits optimal ($O(n)$) und es können sich daher keine Vorteile durch die Ausführungsreihenfolge bei der nebenläufigen Ausführung ergeben. Die Testergebnisse sind in Abb. 7.6 dargestellt.

In Abb. 7.6 sind keine nennenswerten Veränderungen der Laufzeit bei den

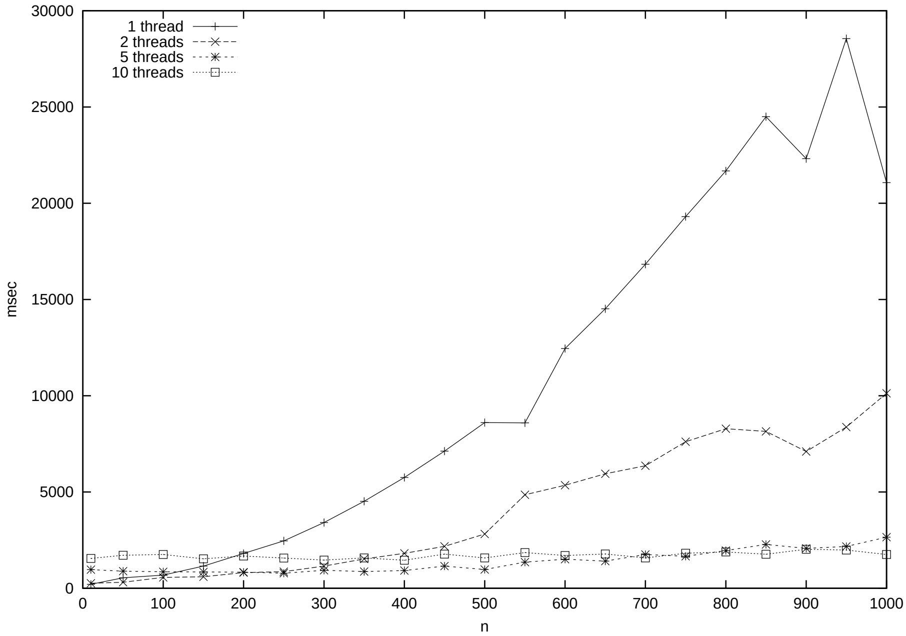

Abbildung 7.5: Nebenläufige Ausführung des *order* Problems in m threads mit jeweils n/m Variablen. Als Aufgabenpuffer in den m lokalen Solvern wurde jeweils ein Stack verwendet. Die Werte sind Durchschnittswerte aus fünf Testläufen, die Varianz der Werte war größer als bei der monolithischen Ausführung in Abb. 7.1

einzelnen Konfigurationen zu erkennen. Hier zeigt sich, daß die Kommunikation über RMI bei mehreren Agenten die Programmausführung insgesamt um einen konstanten Faktor verlangsamt.

Die gleichen Untersuchungen habe ich mit echten Prozessen anstatt *threads* durchgeführt. Hierzu wurde ein zentraler Server gestartet, der für jeden Agenten einen Prozeß erzeugt. Die Mitteilung der Referenzen von *remote* Variablen zwischen den Agenten wurde über den zentralen Server implementiert. Die Programme selbst können dann innerhalb der Agenten ausgeführt werden. Wenn die Domänen aller Variablen nur noch genau ein Element enthalten, beenden sich die Prozesse selbst. Der Server-Prozeß wartet auf die Terminierung aller Agenten und kann somit die Ausführungszeit zur Lösung des Problems ermitteln, die in Abb. 7.7 dargestellt ist.

Um die Performanzvorteile, die sich durch die Problemzerlegung ergeben zu relativieren, habe ich den Test analog zur nebenläufigen Version auch mit dem optimalen Puffer durchgeführt. Die Ergebnisse dieser Untersuchung sind in Abb. 7.8 dargestellt.

In beiden Untersuchungen mit Prozessen zeigt sich, daß der Organisationsaufwand zur echten verteilten Ausführung sehr groß ist. Dieser kann wohl nur bei sehr großen Problemen und bei mehreren Rechnern ausgeglichen werden. In den Untersuchungen wurde nur ein Computer verwendet, so daß effektiv durch die Verteilung keine zusätzliche Rechenzeit zur Verfügung stand. Dennoch konnten das Problem und der Löseprozeß direkt zerlegt und auf Prozesse mit separatem

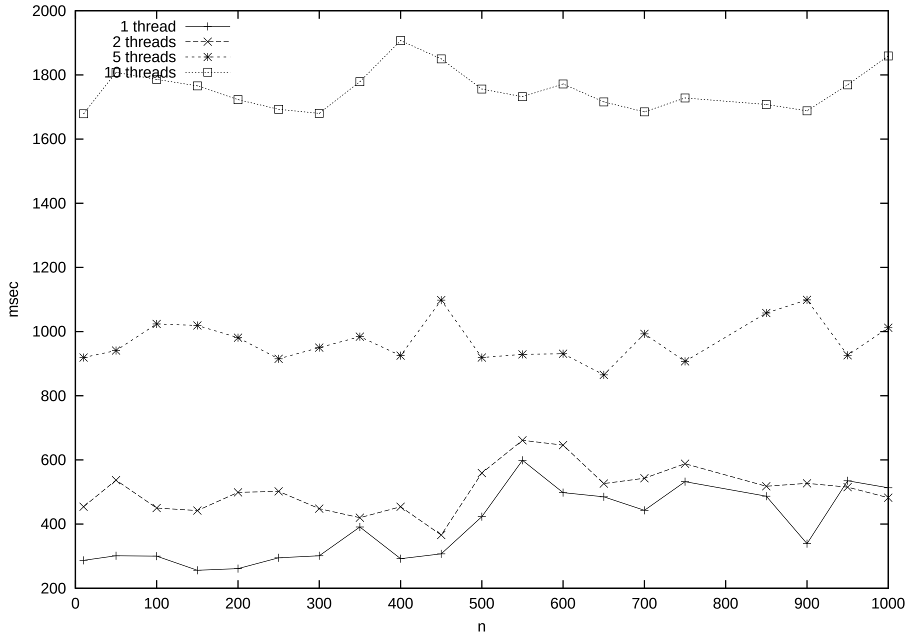

Abbildung 7.6: Nebenläufige Ausführung des *order* Problems in m threads mit jeweils n/m Variablen. Als Aufgabenpuffer in den m lokalen Solvern wurde jeweils der optimale Puffer verwendet, der im vorherigen Abschnitt beschrieben wurde. Die Werte sind Durchschnittswerte aus fünf Testläufen, die Varianz der Werte war größer als bei der monolithischen Ausführung in Abb. 7.1

Speicher verteilt werden, was die Bearbeitung von Problemen erlaubt, die aus Gründen der Speicherkapazität nicht monolithisch gelöst werden können.

7.2 Suchalgorithmen

Zusammenfassung. In diesem Abschnitt werden einige *benchmark*-Probleme angegeben und die vorliegenden Suchconstraints von J.CP damit getestet. Dabei hat sich wie erwartet gezeigt, daß die nicht-deterministischen Algorithmen bei Problemen mit relativ vielen Lösungen schneller sind, aber eine größere Streuung bzgl. ihrer Laufzeit aufweisen.

In der prototypischen ACS Implementierung J.CP stehen wie in Abschnitt 5.4 beschrieben, *backtracking* und *simulated annealing* als generische Suchconstraints zur Verfügung. Der Algorithmus *backtracking* kann außerdem mit unterschiedlichen Wertauswahlheuristiken parametrisiert werden. Diese Algorithmen, obwohl sie nur als *proof-of-concept* gedacht sind, werden in diesem Abschnitt bzgl. ihrer Laufzeit untersucht. Ich habe dazu einige klassische *benchmark*-Probleme implementiert und sie mit J.CP und den verschiedenen Suchalgorithmen lösen lassen. Dabei habe ich die absolute Laufzeit und die Varianz der unterschiedlichen Messungen untersucht. Die Ergebnisse werden in Tabelle 7.1 dargestellt. Die

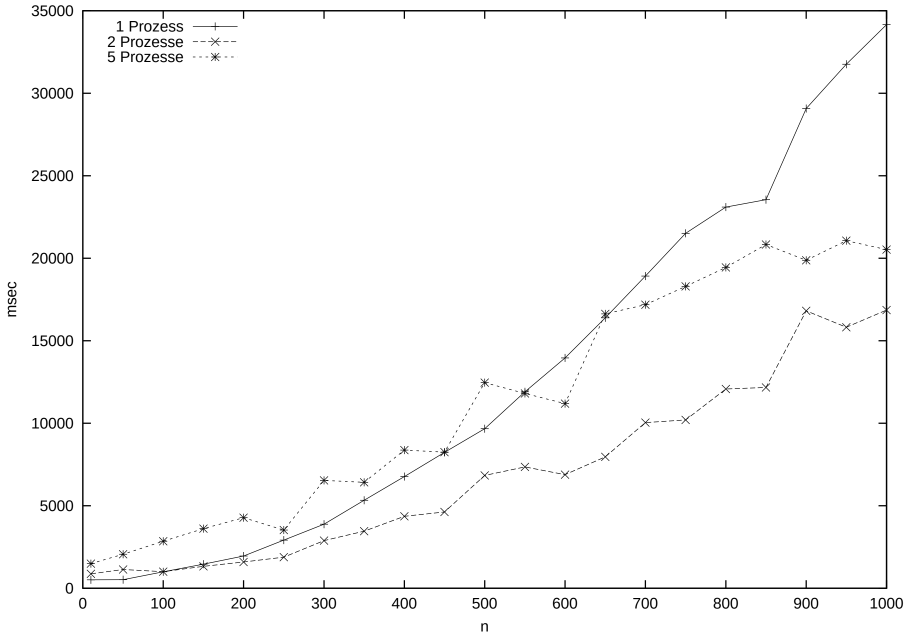

Abbildung 7.7: Verteilte Ausführung des *order* Problems in m echten separaten Prozessen auf einem Computer. Jeder Prozeß behandelt n/m Variablen und kommuniziert mit anderen Prozessen durch die Java RMI Technologie. In jedem Agent läuft ein Solver, der einen Stack als Aufgabenpuffer verwendet. Die Werte sind Durchschnittswerte aus fünf Testläufen, die Varianz der Werte war größer als bei der monolithischen Ausführung in Abb. 7.1

verschiedenen Probleme, die der internationalen *benchmark*-Datenbank CSPlib [GW] entnommen wurden, sind wie folgt definiert:

n-queens: Es sollen n Damen auf einem $n \times n$ großem Schachbrett so verteilt werden, daß sie sich gegenseitig nicht schlagen. Das Problem wird durch n finite-domain Variablen modelliert, die jeweils eine Spalte repräsentieren und deren Wert die Zeile angibt, in der die Dame dieser Spalte steht. Das Problem läßt sich mit n fd-Variablen, $n * (n-1)/2$ Ungleichungen zwischen diesen Variablen und $n * (n-1)$ Ungleichungen mit zusätzlichem Summanden modellieren. Durch Propagation alleine kann hier keine Domäneneinschränkung abgeleitet werden. Der Quotient von Lösungen pro mögliche Variablenbelegungen hängt bei diesem Problem von n ab. Bei $n = 8$ ist er $92/8^8$, bei $n = 100$ lassen sich mit heutigen Werkzeugen nicht alle Lösungen aus den 100^{100} möglichen Variablenbelegungen berechnen.

traffic lights: Bei diesem Problem aus [How95] wird eine Ampelkreuzung mit jeweils vier Ampeln v_1, v_2, v_3, v_4 für Autos und Fußgänger p_1, p_2, p_3, p_4 betrachtet. Die Autoampeln können die diskreten Werte {rot,rot-gelb, grün,gelb} und die Fußgängerampeln die Werte {rot, grün} annehmen. Die 4-ären Constraints über den Variablen v_i und p_i erlauben für die Tupel (v_i, p_i, v_j, p_j) mit $1 \leq i \leq 4, j = (i+1)$ mod 4 jeweils genau 4 verschiedene

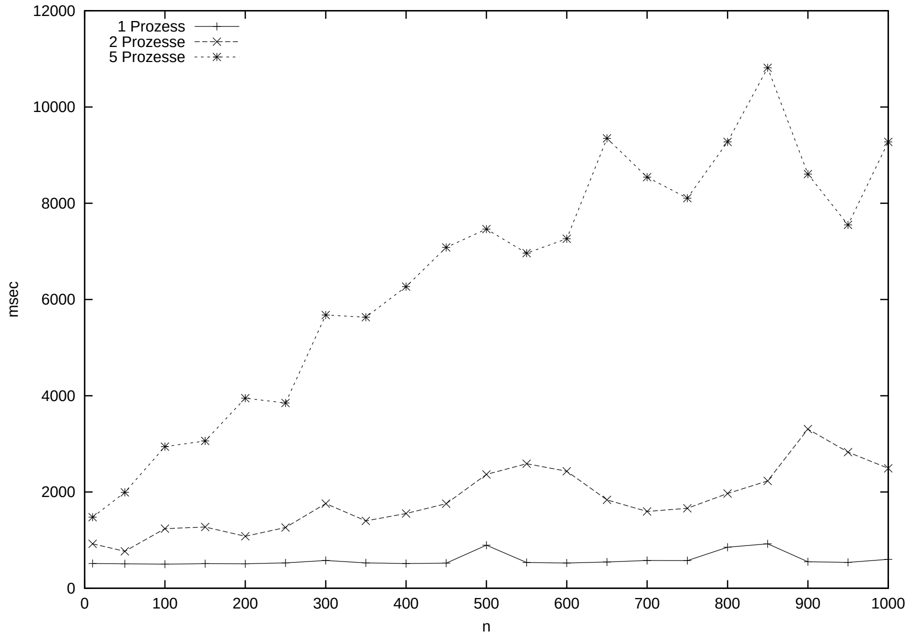

Abbildung 7.8: Verteilte Ausführung des *order* Problems in m echten separaten Prozessen auf einem Computer. Die Solver verwenden den optimalen Puffer für dieses Problem. Die Werte sind Durchschnittswerte aus fünf Testläufen, die Varianz der Werte war größer als bei der monolithischen Ausführung in Abb. 7.1

Belegungen, die explizit bekannt sind. Gesucht sind dann Variablenbelegungen für alle Variablen, so daß eine zulässige Schaltung der Ampeln erzeugt wird. Für dieses Problem gibt es 4 Lösungen aus 2^{12} möglichen Variablenbelegungen.

send: Das Problem besteht darin, den Variablen $\{s, e, n, d, m, o, r, y\}$ Werte zwischen 0 und 9 zuzuweisen, so daß diese Zuweisung eine gültige Summe der Form $send + more = money$ ergibt. Dieses Problem kann direkt mit 8 finite-domain Variablen modelliert und gelöst werden, indem die Summe als Constraint angegeben wird. Für dieses Problem gibt es nur genau eine Lösung aus $9^2 * 10^6$ möglichen Variablenbelegungen, wenn man zusätzlich $s, m \neq 0$ fordert.

golomb n m: Dieses Problem kann mit n Variablen a_i modelliert werden, die aufsteigende Werte haben müssen: $a_1 < a_2 < ... < a_n$, wobei gilt: $a_i \in \{0..m\}$. Die Constraints bestehen darin, daß die $n * (n-1)/2$ Differenzen $a_j - a_i$, mit $1 \leq i < j \leq n$ alle unterschiedliche Werte haben. Mit $n = m^2$ und $m = 5$ gibt es für dieses Problem 132, für $m = 8$ 70349 Lösungen aus den annähernd m^n möglichen Variablenbelegungen.

Der *backtracking*-Algorithmus mit deterministischer Wertauswahl, egal ob aufsteigend oder absteigend, ist bei den Problemen mit relativ vielen Lösungen langsamer als die anderen Algorithmen. Dies wird anhand des n-Damen Problems auch noch mal in den Abbildungen 7.9 und 7.10 dargestellt. Allerdings

Tabelle 7.1: Durchschnittliche Laufzeitergebnisse der verschiedenen Suchalgorithmen von J.CP anhand einiger *benchmark*-Programme. Der erste Wert ist die durchschnittliche gemessene Dauer $\mu(X)$ aus 10 Tests, der Wert in Klammern beschreibt die durchschnittliche prozentuale Standardabweichung $(100 * s(X)/\mu(X))$ der jeweiligen Versuchsreihe.

Problem	BT aufsteigend	BT absteigend	BT zufällig	SA
8-Queens	1008 (7)	899(3)	224 (23)	109(9)
100-Queens	$>$ 3 min	$>$ 3 min	31954 (47)	12342(41)
traffic lights	259(4)	254 (4)	311(4)	235(34)
send	815 (3)	465 (4)	320 (13)	1034 (75)
golomb 5 30	255 (12)	271 (5)	437 (34)	282(53)
golomb 5 10	384 (6)	728 (21)	397 (43)	1875 (138)
golomb 8 50	62344 (19)	54666 (7)	41754 (53)	23453 (85)

unterscheidet sich die Ausführungsdauer, bedingt durch weniger Nichtdeterminismus im Programmablauf, geringfügiger als bei den anderen Algorithmen. Bei Problemen mit relativ wenigen Lösungen im Verhältnis zur Anzahl aller möglichen Variablenbelegungen (Elemente des Suchraums) sind die explorativen *backtracking*-Algorithmen besser als der lokale Suchalgorithmus, weil keine Variablenbelegungen mehrfach untersucht werden. Mit sinkender Anzahl von Lösungen wird die Wahrscheinlichkeit, das eine Lösung durch *simulated annealing* mehrmals generiert wird, größer.

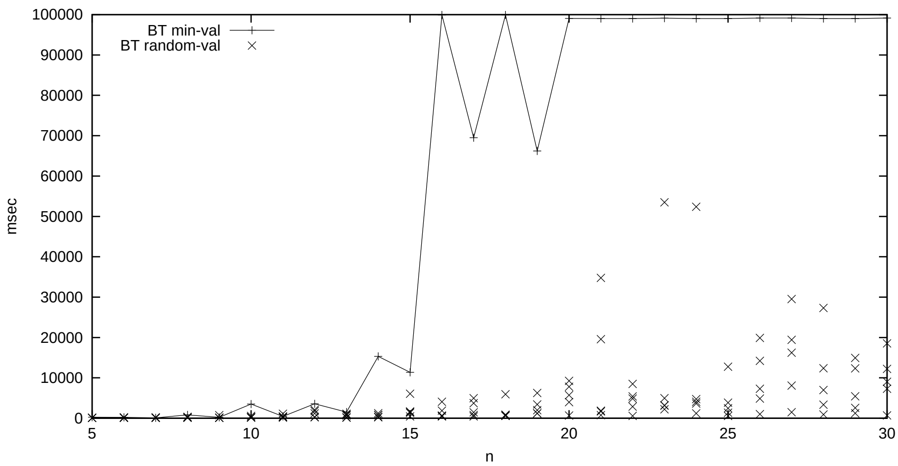

Abbildung 7.9: n-Damen mit Backtracking und zufälliger Wertauswahl im Vergleich zu min-Val. Die Tests wurden nach 1,5 Minuten abgebrochen, so daß bei allen Werten über 90000 davon auszugehen ist, daß die Berechnung noch länger dauert.

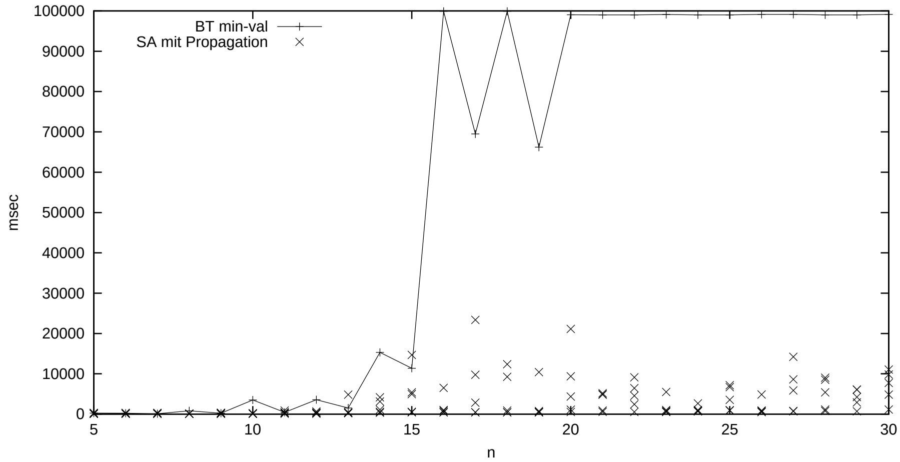

Abbildung 7.10: n-Damen mit Simulated Anealing im Vergleich zu min-Val. Die Tests wurden nach 1,5 Minuten abgebrochen, so daß bei allen Werten über 90000 davon auszugehen ist, daß die Berechnung noch länger dauert.

7.3 Vergleich mit anderen Systemen

Zusammenfassung. In diesem Abschnitt wird J.CP mit anderen Constraintbibliotheken oder Paketen für Java verglichen. Ich betrachte dazu, wie die verschiedenen Systeme die anfangs beschriebenen Leistungsmerkmale erfüllen und gebe ihre Laufzeit zur Lösung einiger klassischer *benchmark* Probleme an. J.CP erfüllt (meine) Leistungsmerkmale am besten von allen Systemen und hat relative gute Laufzeitergebnisse. Nur das professionelle SICStus System findet meist schneller Lösungen als J.CP, die frei erhältlichen Werkzeuge sind alle langsamer.

In Abbildung 7.11 wird ein tabellarischer Überblick über einige Systeme gegeben, die man zur Zeit als Constraintlöser-Bibliothek für die Sprache Java benutzen kann. Die Funktionalität dieser Systeme wird anhand der in Abschnitt 3.1 angegebenen Leistungsmerkmale klassifiziert.

Die Performanz einiger dieser Systeme und J.CP bei der Lösung klassischer *benchmark*-Probleme, die in den vorherigen Abschnitten definiert wurden, ist in Tabelle 7.2 dargestellt. Es wird jeweils die durchschnittliche Laufzeit aus mehreren Tests angegeben. Weil manche der Systeme bestimmte Constraints oder Funktionalitäten nicht zur Verfügung stellen, wurden einige Tests bei manchen Systemen nicht durchgeführt.

Die Solver firstcs und DJCHR wurden gar nicht betrachtet, weil hier noch keine offiziellen Versionen erhältlich sind. Minerva konnte aus technischen Gründen nicht untersucht werden. Aufgrund meiner Erfahrung aus der Entwicklung eines *finite domain* Pakets für diesen Prolog Dialekt [RWG00], schätze ich die Performanz von Minerva aber deutlich schlechter ein, als die von SICStus Prolog.

Die Java Constraint Library (JCL) von der EPFL in Lausanne [JCL], der auf Seite 26 vorgestellt wurde, unterstützt nur unäre und binäre Constraints, so daß die Probleme `send` und `golomb` dort nicht direkt implementierbar waren.

System	Inkrementalität	Adaptivität	Asynchronität	Erweiterbarkeit	Kapselung
SICStus v3.8.7 und jasper [Jas]	Beim Absetzen von Constraints	Durch speichern der Programmabarbeitung, Backtracking und Wiederausführung des entspr. Teilprogramms	Nein, nur ein *thread* erlaubt	Ja, durch Erweiterung des Prolog mit Bibliotheken oder durch CHR	Ja, Prolog-Bibliotheken, allerdings keine gekapselte Suche
firstcs [Fir]	Beim Absetzen von Constraints	Wie bei SICStus	Nein, nur ein *thread* erlaubt	Ja, teilweise auch durch *interfaces* unterstützt	Nein.
JCL v2.1 [JCL]	Nein	Nein, aber leicht integrierbar weil nicht inkrementell	Nein	Ja, aber nur mit Insider-Wissen	Ja, CSP-Modelle und Suche sind einzelne Komponenten
JaCK [JaCK]	Ja	Nein	Nein	Ja, durch neue CHR-Regeln	Ja, Constraintsysteme durch CHR Programme und Suche durch JASE
DJCHR [Wol01]	Ja	Ja	Nein	Ja, durch neue CHR-Regeln	Suchalgorithmen für FD austauschbar
Minerva [Min]	Muß für jedes Constraint implementiert werden	Nein	Ja, verwendet Minerva *sockets* zur Verteilung	Ja, aber aufwendig. Constraints müssen implementiert werden	Nein, alle Constraints sind Prädikate, Suche systeminhärent
J.CP	Ja	Ja	Ja	Ja, durch *interfaces* unterstützt	Ja, Pakete für Constraintsysteme, Suchconstraints für Suchalgorithmen

Abbildung 7.11: Vergleich der Funktionalität von Java Paketen zur Constraintprogrammierung anhand der Leistungsmerkmale aus Abschnitt 3.1

Die Modellierung der mehrstelligen Constraints durch binäre wie es in [RPD90] beschrieben wird, habe ich für das `traffic` Problem angewandt. Dieses Problem läßt sich besonders gut in JCL eingeben, weil dort die Constraints explizit aufgezählt werden müssen und es hier nur sehr wenige gibt. Das `reorder` Problem läßt sich mit JCL nicht bearbeiten, weil es keine Möglichkeit gibt, Constraints aus einem Modell zu entfernen, obwohl dies bei einem nicht-inkrementellen System wie JCL sehr einfach wäre.

Tabelle 7.2: Vergleich der Laufzeiten einiger Bibliotheken zur Constraintverarbeitung in Java anhand bekannter *benchmark*-Probleme. Die Probleme, deren Felder mit X markiert sind, konnten nicht innerhalb von 3 Minuten gelöst werden. Tests, die in mit – markierten Feldern stehen, wurden nicht durchgeführt.

	J.CP BT	J.CP SA	JCL	JaCK	jasper	jasper ff
`8 queens`	324	312	23	1061	768	760
`20 queens`	3617	556	18575	X	12228	1061
`50 queens`	9326	2015	X	X	X	4968
`100 queens`	20934	12342	X	X	X	21231
`traffic`	311	235	476	–	–	–
`send`	320	1034	–	X	1074	993
`golomb 5 30`	437	282	–	–	1032	752
`golomb 5 10`	397	1875	–	–	1011	738
`golomb 8 50`	41754	23453	–	–	7928	9243
`golomb 10 200`	X	370321	–	–	549158	576709
`order 100`	703	712	78366	90366	809	804
`order 1000`	13124	13332	X	X	6934	6938
`reorder 1000 500`	6173	6322	–	–	7389	7356

In dem CLP System SICStus Prolog, dessen Ausführungsmodell auf Seite 23 beschrieben wurde, habe ich das `traffic` Problem nicht implementiert, weil hier keine symbolischen Constraintdomänen unterstützt werden. Es wäre zwar eine Modellierung durch Zahlen möglich, dies habe ich aber nicht durchgeführt, weil dieses Problem gerade wegen der symbolischen Domänen ausgewählt wurde. Die Constraintrücknahme ist in SICStus durch *backtracking* bis zu dem Punkt vor dem zurückzunehmenden Constraint und anschließender Neuberechnung aller darauf folgenden Constraints implementiert. Es wird also der letzte Zustand, der ohne das zurückzunehmende Constraint existierte, wiederhergestellt. Dann werden alle Constraints, die nach dem zurückzunehmenden abgesetzt wurden, nochmals abgesetzt. Dies ist in Prolog bzw. mit Hilfe der Java-Schnittstelle jasper möglich, wenn man a priori weiß, welches Constraint später ggf. zurückgenommen werden soll.

Die Java Bibliothek JaCK zur Constraintverarbeitung mit CHR wurde mit dem in der Kontribution enthaltenen Solver für *finite domain* Constraintverarbeitung für `send` und `order` verwendet. Das Ausführungsmodell von CHR, das hier zur Constraintverarbeitung genutzt wurde, wird auf Seite 24 eingeführt. Für `queens` ist ein spezieller Löser definiert, der für die Tests verwendet wurde. Das `reorder`-Problem konnte nicht gelöst werden, weil die Constraintrücknahme für CHR [Wol99] in JaCK nicht verwendet wird. Da in den vorhandenen Constrainthandlern keine Constraints zur Verarbeitung von Differenzen vorhanden waren,konnte `golomb` nicht implementiert werden. Für `traffic` müßten die speziellen symbolischen Constraints zunächst implementiert werden, so daß ich auch diesen Test nicht durchgeführt habe.

Kapitel 8

Ergebnisse und Ausblick

8.1 Ergebnisse

Mit ACS wurde ein domänenunabhängiges Ausführungsmodell zur inkrementellen Constraintverarbeitung definiert. Domänenspezifische Constraints und Variablen können als gekapselte Pakete integriert werden, wodurch Constraintlöser für konkrete *constraint satisfaction* Probleme erzeugt werden. Die Verarbeitung neuen Wissens in solchen Constraintlösern erfolgt inkrementell, d.h. jede neue Information wird sofort möglichst weitgehend mit dem vorhandenen Wissen verknüpft, ohne dazu alle vorherigen Berechnungen erneut auszuführen.

Im Unterschied zu den meisten herkömmlichen Constraintlösern können in ACS nicht nur neue Constraints in die Wissensbasis aufgenommen, sondern auch vorhandene Constraints entfernt werden. Die Ausführung dieser beiden Operationen erfolgt dabei asynchron zum aufrufenden ACS-Constraintprogramm. Desweiteren unterscheidet sich ACS von anderen Systemen dadurch, daß mehrere Constraintlöser kooperativ und nebenläufig gemeinsame Constraintprobleme verarbeiten können, ohne dabei zusätzlichen Koordinationsaufwand zu erzeugen.

Durch die Asynchronität und Nebenläufigkeit ist die Definition oder Verwaltung globaler Systemzustände in ACS kaum möglich. In den meisten etablierten Systemen zur Constraintverarbeitung erfüllen aber gerade solche Zustände zur Umsetzung nicht-deterministischer Suchalgorithmen, wie z.B. *backtracking*, eine zentrale Aufgabe im Ausführungsmodell. Nicht-deterministische Suche muß in den meisten Anwendungsbereichen der Constraintprogrammierung eingesetzt werden, weil die behandelten *constraint satisfaction* Probleme i.a. NP-vollständig sind. In ACS können alternative Variablenbelegungen während der Suche erzeugt und ggf. verworfen werden, indem sie als Instantiierungen und damit als Constraints abgesetzt und bei Mißerfolg zurückgenommen werden. Dadurch kann z.B. auch *backtracking* implementiert werden: Anstatt ehemalige Zustände aus einem zentralen Speicher zu laden, werden falsche Entscheidungen direkt durch die entsprechende Umkehroperation der Instantiierung, nämlich durch Rücknahme des Instantiierungsconstraints, rückgängig gemacht.

Aus jedem ACS-Constraint kann eine Propagationsfunktion abgeleitet werden, die die Domänen seiner Variablen in Abhängigkeit von den Domänen anderer Variablen gemäß dem implementierten Expertenwissen über die Constraintdomäne einschränkt. Die Definitions- und Wertebereiche dieser Funktionen wurden in dieser Arbeit kanonisch so erweitert, daß alle Funktionen auf einer gemeinsamen Menge, nämlich dem Kreuzprodukt aller Variablendomänen, operieren. Damit kann die Theorie der Chaotischen Iteration zur Berechnung eines eindeutigen Fixpunktes aus den Propagationsfunktionen aller abgesetzten Constraints verwendet werden. Dieser Fixpunkt ist erreicht, wenn sich durch keine dieser Propagationsfunktionen weitere Domäneneinschränkungen ableiten lassen. Somit kann der Fixpunkt als die denotationale Semantik eines Constraintprogramms betrachtet werden.

Das ACS Ausführungsmodell berechnet diesen Fixpunkt, unterscheidet sich aber in drei wesentlichen Punkten von der Chaotischen Iteration und wird erst dadurch praktisch anwendbar:

- Die Berechnung des Fixpunktes erfolgt in endlicher Zeit, wenn er existiert und die Propagationsfunktionen selbst endliche Berechnungen durchführen.
- In den Propagationsfunktionen können weitere Constraints abgesetzt werden, was die Implementierung globaler Constraints wesentlich unterstützt.
- Die Iteration kann unterbrochen und später fortgeführt werden, was die Integration der Constraintrücknahme erlaubt.

Durch das Zurücknehmen von Constraints können Variablendomänen erweitert werden. Deswegen kann die funktionale Interpretation dieser Operation auf dem Kreuzprodukt der Variablendomänen nicht in die Menge der Funktionen zur Berechnung des Fixpunktes der Chaotischen Iteration aufgenommen werden. Existenz und Eindeutigkeit des Fixpunktes wären dann nicht mehr gesichert.

In dieser Arbeit wurde die Constraintrücknahme als Operation in den chaotischen Iterationsprozeß der Propagation von ACS-Constraints integriert. Die intendierte Semantik der Operation, nämlich einen Zustand zu erzeugen, der entstanden wäre, wenn das zurückgenommene Constraint nie abgesetzt worden wäre, konnte dabei formal und praktisch umgesetzt werden. Zur Ausführung der Constraintrücknahme wird der chaotische Iterationsprozeß der Propagation dadurch unterbrochen, daß die Rücknahmeoperation in allen beteiligten Constraintlösern ausgeführt wird. Es werden dann die Domänen aller möglicherweise betroffenen Variablen erweitert und anschließend die Chaotische Iteration durch erneutes Absetzen relevanter Constraints wieder angestoßen. Dadurch wird schließlich der Fixpunkt berechnet, der ohne das zurückgenommene Constraint theoretisch als Ergebnis der Chaotischen Iteration definiert ist.

Durch die generische Konzeption des Ausführungsmodells ACS können in seiner Objekt-orientierten Implementierung J.CP viele verschiedene Arten von Constraintproblemen behandelt werden, wenn entsprechende Constraints zur Verfügung stehen. Heterogene Constraintprobleme, deren Constraints sich auf Variablen verschiedener Constraintdomänen beziehen, können ohne weitere Kooperationseinheiten gelöst werden. Zur Implementierung heterogener Constraints wird die strenge Typisierung der Implementierungssprache Java und Expertenwissen über mögliche Zusammenhänge zwischen Werten unterschiedlicher Constraintdomänen benötigt. Die nach Constraintdomänen getrennte Bearbeitung von Teilproblemen und die komplexe Kombination von deren Lösung ist mit J.CP nicht mehr notwendig.

Auch verteilte Anwendungen können mit J.CP ad hoc implementiert werden, wenn die entsprechenden Kommunikationsmethoden in den Constraints zur Verfügung stehen. Zur Verarbeitung von Constraints über endlichen Mengen und über ganzen Zahlen sind in J.CP bereits Constraints und Variablen zur Bearbeitung verteilter Constraintprobleme implementiert. Das ACS Ausführungsmodell erlaubt die Anwendung in verteilten Systemen, weil in ihm keinerlei globale Datenstrukturen verwendet werden und ein Großteil der notwendigen Kommunikation asynchron ausgeführt wird. Um die Korrektheit und Vollständigkeit des Ausführungsmodells bei Verteilung zu erhalten, muß nur die Berechnung der Zuverlässigkeit und die Unterbrechung der Propagation bei Constraintrücknahme synchronisiert werden. Die Möglichkeit, jedes Constraint und damit auch jede Instantiierung zurückzunehmen, erlaubt die Integration beliebiger Suchalgorithmen zur Problemlösung. Jede nicht-deterministisch getroffene Entscheidung zur Instantiierung einer Variable kann unabhängig vom Zeitpunkt oder Ort in dem sie abgesetzt wurde, zurückgenommen werden. Damit sind auch lokale Suchverfahren, die lokale Änderungen durch Constraintrücknahme und erneuter Instantiierung durchführen, in J.CP integrierbar. In der vorliegenden Implementierung wurden zwei generische, d.h. für die meisten bekannten Constraintdomänen einsetzbare, Suchalgorithmen umgesetzt. Ein vollständiger *backtracking*-Algorithmus und ein unvollständiger lokaler Suchalgorithmus (*simulated annealing*) können für gegebene Constraintprobleme beliebig eingesetzt und ausgetauscht werden.

Die vorhandene prototypische ACS Implementierung J.CP kann bzgl. Performanz und Speicherbedarf bereits für herkömmliche *benchmark* Probleme eingesetzt werden. Die Effizienz ist hier vergleichbar mit der anderer Constraintlöser, die als Java Bibliotheken verwendet werden können, wobei aber J.CP, wie beschrieben, mehr Funktionalität zur Verfügung stellt. Durch die asynchrone Ausführung der Propagationsfunktionen ergeben sich Performanzgewinne im Vergleich zur synchronen Ausführung von Constraintprogrammen. Die Priorisierung von Constraints erlaubt es, später hinzugekommenes, wichtiges Wissen bevorzugt zu verarbeiten, so daß sich insgesamt der Rechenaufwand reduziert. Die Constraintrücknahme sollte z.B. immer möglichst früh ausgeführt werden, damit das zurückzunehmende Constraint möglichst wenig Veränderungen veranlaßt hat, die dann rückgängig gemacht werden müssen. Die generischen und gekapselten Suchalgorithmen ermöglichen die individuelle Auswahl des Algorithmus für das vorliegende Problem. Bei Constraintproblemen mit vielen Lösungen sollte z.B. ein lokaler Suchalgorithmus eingesetzt werden, weil er hier sehr schnell eine Lösung findet. Wenn alle Lösungen eines Problems gesucht sind, muß der vollständige *backtracking*-Algorithmus verwendet werden. Für weitere spezielle Anwendungsfelder lassen sich leicht weitere, spezialisierte Suchalgorithmen integrieren, die, je nach gewünschten Eigenschaften, aus der Literatur ausgewählt werden können.

Durch das Ausführungsmodell ACS und seine Implementierung eröffnen sich neue Anwendungsfelder der Constraintprogrammierung. Mit der asynchronen und adaptiven Verarbeitung von Constraints können besonders gut interaktive und verteilte Systeme implementiert werden. Die die Arbeit begleitende Fallstudie zur verteilten Terminplanung ist zum Beispiel mit herkömmlichen Constraintlösern kaum zu bearbeiten. Alle bekannten Constraintlöser außer Mozart sind nicht in der Lage, entfernte Variablen zu verarbeiten, so daß immer unterschiedliche Kopien der Constraintvariablen verwaltet und zwischen den Agenten synchronisiert werden müssen. In ACS bzw. mit J.CP kann man direkt auf die Variablen zugreifen, ohne dadurch das System durch den erhöhten Kommunikationsaufwand erheblich zu verlangsamen. In Mozart muß die Propagation über Prozessgrenzen hinweg dagegen aus Performanzgründen eingeschränkt werden, weil die Propagation dort synchron erfogt.

Bei der verteilten Terminplanung sollten bereits eingeplante Termine verschoben oder abgesagt werden können. Wenn es hierfür keine entsprechende Operation gibt, wie z.B. in CLP oder Mozart, muß ein ehemaliger Zustand des gesamten Systems konstruiert werden um dort den neuen Zustand zu berechnen. Dies ist aber bei der kontinuierlichen, offenen und verteilten Anwendung aus der Fallstudie nicht möglich, weil man die Zustände aller laufenden Systeme nicht zeitlich synchronisieren kann. Die Operation zur Constraintrücknahme aus ACS berechnet den gewünschten Zustand inkrementell aus lokal gespeicherten Daten und ist somit im Gegensatz zu allen anderen Systemen auch für dieses Problem in der vorliegenden Modellierung anwendbar.

8.2 Ausblick

Durch die detaillierte Definition des Ausführungsmodells ACS sind genaue Vorgaben an Eigenschaften und Verhalten der Constraints und Variablen für ihre Implementierungen festgelegt. Die vorgestellten Vollständigkeits- und Korrektheitseigenschaften der durch konkrete Implementierungen entstehenden Constraintlöser sind damit in jedem Fall gesichert. Daher kann ACS als Ausgangspunkt zur Erstellung von Constraintlösern durch die Implementierung konkreter Constraints und Constraintsysteme (also Variablendomänen) betrachtet werden. Dies wurde bereits an den bekannten Constraintdomänen der endlichen Mengen und der ganzen Zahlen für die ACS Implementierung J.CP als *proof-of-concept* durchgeführt. In diese vorhandenen Constraintsysteme sollten, um den Solver J.CP für alle bekannten Anwendungsbereiche dieser Constraintdomänen verwendbar zu machen, weitere bekannte Constraints (z.B. *cumulative, alldifferent,...*) integriert werden. Dazu müßten die entsprechenden Algorithmen aus der Literatur als Propagationsmethoden von entsprechenden Constraint-Klassen implementiert werden.

Weitere bekannte Constraintsysteme können, um die Anwendungsmöglichkeiten von J.CP zu erweitern, als Java-Pakete analog zu den beiden vorhandenen Paketen integriert werden. Reellwertige Variablendomänen mit arithmetischen Constraints sind zum Beispiel eine klassische und erfolgreich angewandte Constraintdomäne, die insbesondere in der Kombination mit Ganzzahlarithmetik zur Modellierung vielfältiger Anwendungen der Ingenieur- und Wirtschaftswissenschaft geeignet ist. Für solche speziellen Constraintdomänen existieren oft besonders effiziente Lösungsalgorithmen (z.B. *simplex*), die meist in konkreten Anwendungsbereichen der Wirtschaftswissenschaft (*operations research*) entwickelt wurden. Um solche domänenspezifischen Algorithmen für das generische System J.CP zu nutzen, müßten sie als globales Constraint als Teil eines entsprechenden Constraintsystems integriert werden. Dies erfordert die Abgrenzung der speziellen Algorithmen von der Propagation und Inkonsistenzverarbeitung von ACS. Diese beiden Eigenschaften können aber in ACS individuell für jedes Constraint definiert werden, so daß eine Integration theoretisch immer möglich wäre. Zu klären bleibt im Einzelfall, ob sich die Constraintsysteme wirklich verbinden lassen, ob es also eine geeignete Interpretation des einen Systems im anderen gibt, mit der sich die berechneten Ergebnisse darstellen lassen. Mit einer solchen

Integration von Constraintsystemen würde J.CP nicht nur die Verarbeitung von Variablen unterschiedlicher Constraintdomänen, sondern auch die Kooperation unterschiedlicher Konsistenz- bzw. Suchalgorithmen zum Lösen von CSPs unterstützen.

Durch die Kapselung der Suchalgorithmen als Constraints in ACS können beliebige Algorithmen für J.CP adaptiert werden. Dazu muß lediglich das Verhalten des Algorithmus sowohl durch eine Einheit zur Wert- bzw. Variablenauswahl als auch durch eine die auf Inkonsistenz-Ereignisse reagiert, beschrieben werden. An zwei bekannten Algorithmen wurde dies in der ACS Implementierung J.CP bereits durchgeführt. Es lassen sich noch weitere Algorithmen bzw. effizientere Suchstrategien für die vorhandenen Algorithmen definieren. Die *first fail* Strategie zum Beispiel, die die Variablenauswahlfunktion während der Suche dynamisch verändert, würde in sehr vielen Anwendungen von *backtracking* erhebliche Performanzgewinne bringen. Oder ein "echter" verteilter Suchalgorithmus, wie *asynchronous backtracking*, könnte als globales Suchconstraint in offenen verteilten Systemen implementiert werden, um den gesamten vorhandenen Suchraum behandeln zu können. Allerdings ist auch für solche Algorithmen immer eine zentrale Instanz nötig, die den Algorithmus startet und seine Termination feststellt. Sehr viel Potential liegt sicherlich auch in der Implementierung verteilter lokaler Suchalgorithmen, weil lokale Reparaturen dabei in den Agenten durchgeführt werden könnten und somit weniger globale Abstimmung notwendig wird.

In offenen verteilten Systemen wird die Termination von verteilten Suchalgorithmen, wie *asynchronous backtracking*, in der Literatur bisher nur mit allgemeinen Methoden zum Erkennen global stabiler Zustände durchgeführt. Die Autoren geben an, daß man einen Algorithmus, der einen global stabilen Zustand erkennt, dazu nutzen kann, um festzustellen, ob die Suche beendet ist. Das Problem dieser *termination detection* ist, daß solche Zustände in interaktiven verteilten Anwendungen oft gar nicht auftreten, weil die Eingaben unterschiedlicher Agenten gleichzeitig verarbeitet werden. Wenn z.B. in unterschiedlichen Agenten eines verteilten Systems immer wieder unabhängig voneinander Suche auf paarweise disjunkten Constraintnetzen gestartet wird, so wird ein solcher Algorithmus nie einen global stabilen Zustand feststellen können. Die Berechnung einer Lösung, also die Ausführung der Suche, ausgehend von einem der Agenten, kann aber trotzdem abgeschlossen sein. Daher sollten Algorithmen entwickelt werden, die die Termination einer bestimmten Berechnung in einem verteilten System erkennen, auch wenn in der selben Anwendung noch andere Berechnungen laufen. Durch die gespeicherten Abhängigkeiten von Constraints in ACS sind hier sicherlich effiziente, ACS-spezifische Algorithmen zur Berechnung der verteilten Zuverlässigkeit möglich. Die Terminierung, also das Erreichen der höchsten Zuverlässigkeit eines Suchalgorithmus oder auch normaler Propagation, kann mit Hilfe der Abhängigkeiten von Constraints untersucht werden, indem nur die Terminierung der Berechnungen geprüft wird, die durch die Suche angestoßen wurden. Die theoretische Untersuchung solcher Algorithmen und eine Implementierung in Kombination mit einem echten verteilten Suchalgorithmus wäre eine wesentliche Weiterentwicklung von ACS in Richtung verteilte Systeme.

Die vorhandene Implementierung kann, insbesondere für die verteilte Anwendung, bzgl. ihrer Performanz optimiert werden. Die Kommunikation zwischen entfernten Komponenten sollte z.B. für verschiedene Arten der physischen Verbindung individuell angepaßt werden können. Bei schnellen *ethernet* Verbindungen ist es zum Beispiel günstiger, jede Nachricht möglichst schnell zu verschicken, damit die verschickte Information möglichst früh verarbeitet werden kann, während der Verbindungsaufbau im Internet so aufwendig ist, daß es sich dort empfiehlt, größere Nachrichtenpakete zu "packen" und Informationen dafür zu sammeln. Am Ausführungsmodell müssen dazu wegen seiner inhärenten Asynchronität keine Änderungen vorgenommen werden, aber die Kommunikation müßte auch formal neu definiert werden, um das "Packen" von Information zu ermöglichen. In den einzelnen Constraintsystemen können außerdem sicherlich noch Laufzeitoptimierungen durch effizientere Algorithmen in den Constraints und Variablendomänen erreicht werden. Zur internen Darstellung endlicher Mengen ganzer Zahlen zum Beispiel wurden in der Literatur viele Konzepte untersucht, die effizienter sind als die in J.CP gewählte einfache Repräsentation.

Die Fallstudie zur verteilten Terminplanung soll als Einstieg in das praktisch sehr relevante und wissenschaftlich vielseitig bearbeitete Anwendungsgebiet dienen. Die Möglichkeiten von ACS zum asynchronen und verteilten Constraintlösen erlauben, im Vergleich zu anderen Ansätzen, eine sehr gute und realitätsnahe Modellierung. Die vorhandene Implementierung kann als Basis für ein praktisch anwendbares Software-Paket betrachtet werden. Alle wesentlichen Komponenten sind integriert, obwohl sicherlich an vielen Stellen noch Verbesserungen bzw. Erweiterungen notwendig sind. Die Benutzerinteraktion muß z.B. verbessert werden, damit die Software konkurrenzfähig zu kommerziellen *tools* wird. Weitere Suchconstraints und Constraints zur besseren Planung, die in einem größeren Kontext nach optimalen Lösungen suchen, sollten noch integriert werden. Für solche Erweiterungen ist durch die Objekt-orientierte Architektur und die Kapselung von Constraints eine exakte Schnittstelle vorgegeben.

Verzeichnis der Definitionen

Index

Literaturverzeichnis

[AB93] Abderrahamane Aggoun and Nicolas Beldiceanu. Overview of the chip compiler system. In *Constraint Logic Programming: Selected Research*, pages 421–437. MIT Press, 1993.

[AO99] Ken Arnold and Bryan O'Sullivan. *The Jini Specification*. Addison Wesley, 1999.

[Apt98] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1998.

[BDFB+87] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint hierarchies. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 48–60, New York, NY, 1987. ACM Press.

[Bes91] Christian Bessiére. Arc-consistency in dynamic constraint satisfaction problems. In *Proc. AAAI'91*, pages 221–226, 1991.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[BN95] P. Berlandier and B. Neveu. Problem partitition and solvers coordination in distributed constraint satisfaction. In *Proc. PPAI95*, 1995.

[Boo94] Grady Booch. *Object-oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.

[BS95] Franz Baader and Klaus U. Schulz. Combination of constraint solving terchniques: An algrbraic point of view. In *Lecture Notes in Computer Science 914*, pages 352–366, 1995.

[BS98] Franz Baader and Klaus U. Schulz. Combination of constraint solvers for free and quasi-free structures. *Theoretical Computer Science*, 192:107–161, 1998.

[BW98] Martin Buchi and Wolfgang Weck. Compound types for java. In *Proc. OOPSLA '98*, pages 362–373, 1998.

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 1996.

[CD01] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In Kathleen Steinhöfel, editor, *Proc. SAGA 2001*, LNCS 2264, pages 73–89. Springer, 2001.

[CDR96] P. Codognet, D. Diaz, and F. Rossi. Constraint retraction in fd. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 1180. Springer, 1996.

[Chip] Cosytec. Chip 5. `http://www.cosytec.com`.

[CHN01] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In *Proc. CP01*, LNCS 2239, pages 240–255. Springer, 2001.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[CLP] Yan Georget. clp(FD,S) web pages. `http://contraintes.inria.fr/~georget/clp_fds.html`.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1989.

[CLS98] K. Choi, J. Lee, and P. Stuckey. A lagrangian reconstruction of a class of local search methods. In *Proc. 10th Int'l Conf. on Artificial Intelligence Tools*. IEEE Computer Society, 1998.

[CM69] D. Chazan and W. Miranker. Chaotic relaxiation. *Linear Algebra and its Application*, 2:199–222, 1969.

[COC97] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kucken, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *LNCS*, pages 191–206. Springer-Verlag, 1997.

[Conc] ILOG S.A. *ILOG Concert 1.0 User's Manual*, August 2000.

[Col84] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proc. 2nd Int'l. Conf. on Fifth Generation Computer Systems*, pages 85–99, 1984.

[Cou83] Bruno Courdelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.

[CS88] Volker Claus and Andreas Schwill. *Duden Informatik*. Dudenverlag, 1988.

[DD98] R. Dechter and A. Dechter. Belief maintainance in dynamic constraint networks. In *Proc. AAAI'88*, pages 37–42, 1998.

[DH96] Ron Davidson and David Harel. Drawing graphs nicely unsing simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[Doy79] Jon Doyle. A truth maintainance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[Dur99] Edmund H. Durfee. Distributed problem solving and planning. In Gerhard Wei, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 121–164. MIT Press, 1999.

[DvHS+88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS88)*, 1988.

[ecl] IC-Parc at Imperial College London. ECL^{i}PS^e internet pages `http://www.icparc.ic.ac.uk/eclipse/`.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag Berlin Heidelberg, 1985.

[EZR94] Eithan Ephrati, Gilad Zlotkin, and Jeffrey S. Rosenschein. A non–manipulable meeting scheduling system. In *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, Seatle, WA, 1994.

[FA97] Thom Frühwirth and Slim Abdenadher. *Constraint-Programmierung.* Springer, Berlin Heidelberg, 1997. in german.

[FC98] C. Fetzer and F. Cristian. A fail-aware datagram service. In D.R. Avaresky and D.R. Kaeli, editors, *Fault-Tolerant Parallel And Distributed Systems*, chapter 3, pages 55–69. Kluwer Academic Publishers, 1998.

[FCar] C. Fetzer and F. Cristian. Fail-awareness: An approach to construct fail-safe applications. *Journal of Real-Time Systems*, to appear.

[FFS98] F. Fages, J. Fowler, and T. Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37:1-3:185–212, Oct-Dec 1998.

[Fig] National University of Singapore. Figaro library. `http://figaro.comp.nus.edu.sg`.

[Fir] Fraunhofer FIRST. firstcs solver. `http://www.first.fhg.de/~hs/pooc`.

[FK02] F.GLover and G. Kochenberger, editors. *Handbook on Metaheuristics.* Kluwer Academic Publishers, 2002. To be published.

[FLL01] Filippo Focacci, Francois Laburthe, and Andrea Lodi. Local search and constraint programming. In *Tutrial notes at CP 2001*, 2001. To be published also in [FK02].

[FMW01] Eugene Freuder, Marius Minca, and Richard Wallace. Privacy/efficiency tradeoffs in distributed meeting scheduling by constraint-based agents. In *Proc. IJCAI01 Workshop on Distributed Constraint Reasoning*, 2001.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.), Journal of Logic Programming*, 37(1-3):95–138, 1998.

[GCR99] Yan Georget, Philippe Codognet, and Francesca Rossi. Constraint retraction in CLP(FD). *Constraints*, 4(1):5–42, February 1999. ISSN 1383-7133.

[Geo99] Yan Georget. *Constraint Programming reactive extensions.* PhD thesis, cole Polytechnique Paris, 1999.

[Ger97] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[GL93] Fred Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Oxford, England, 1993. Blackwell Scientific Publishing.

[GM00] Hans-Joachim Goltz and Dirk Matzke. On modelling and solving timetabling problems using constraint logic programming. In P. Meseguer and C. Bessiere, editors, *Proc. ECAI-Workshop Modelling and Solving Problems with Constraints*, 2000.

[Gnu] Daniel Diaz. GNUProlog Internet site. `http://gnu-prolog.inria.fr`.

[GS96] Leonardo Garrido and Katia Sycara. Multi-agent meeting scheduling: Prelimenary experimental results. In *Proc. ICMAS96*, 1996.

[GTdW93] F. Glover, E. Taillard, and D. de Werra. Tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[GW] Ian Gent and Toby Walsh. CSPLib, a problem library for constraints, `http://www-users.cs.york.ac.uk/~tw/csplib/`.

[Han01] Markus Hannebauer. *Autonomous Dynamic Reconfiguration in Collaborative Problem Solving.* PhD thesis, TU Berlin, 2001.

[HE80] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14*, 14(3):263–313, 1980.

[Hen89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[HG96] David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press, 1996.

[HH89] Ralf Guido Herrtwich and Günter Hommel. *Kooperation und Konkurrenz*. Springer, 1989.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hof98] Petra Hofstedt. An architecture for the combination of constraint solvers. In *Proceedings of the ERCIM/COMPULOG Workshp on Constraints*, 1998.

[Hof01] Petra Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, TU Berlin, 2001.

[Hol90] Christian Holzbauer. *Specification of Constraint Based Inference Mechanisms through Extended Unification*. PhD thesis, Institut fuer Med.Kybernetik u. AI, Universitaet Wien,, 1990.

[Hol92] Christian Holzbaur. Metastructures vs. attributed variables in the context of extensible unification - applied for the implemetation of clp languages. Technical report, Austrian Research Institute for Artificial Intelligence, 1992.

[How95] Walter Hower. Constraint satisfaction — algorithms and complexity analysis. *Information Processing Letters*, 55(3):171–178, August 1995. Elsevier Science Publishers B.V.

[HS88] M. Höhfeld and Gert Smolka. Definite relations over constraint languages. Technical report, LILOG Report 53,IWBS,IBM, 1988.

[HSG01] P. Hofstedt, D. Seifert, and E Godehardt. A framework for cooperating constraint solvers - a prototypic implementation. In *Proc. CoSolv*, 2001.

[HSW93] Martin Henz, Gert Smolka, and Jörg Würtz. Oz – a programming language for multi-agent systems. In Ruzena Bajcsy, editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 404–409, Chambery, France, 1993.

[HvRBS98] Seif Haridi, Peter van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998. Invited Paper.

[HW96] Martin Henz and Jörg Würtz. Using Oz for college time tabling. In E.K.Burke and P.Ross, editors, *The Practice and Theory of Automated Time Tabling: The Selected Proceedings of the 1st International Conference on the Practice and Theory of Automated Time Tabling, Edinburgh 1995*, Lecture Notes in Computer Science, vol. 1153, pages 162–177, 1996.

[Ilog] ILOG S.A. *ILOG Solver 5.0 User's Manual*, August 2000.

[ILOG] ILOG S.A. `http://www.ilog.fr`.

[JaCK] LMU Munich. Java Constraint Kit JaCK. `http://www.pms.informatik.uni-muenchen.de/software/jack/`

[Java] SUN Microsystems. Java, `http://java.sun.com`.

[Jas] Swedish Institute of Computer Science. Sisctus prolog jasper interface. `http://www.sics.se/sicstus/docs/latest/html/jasper/`.

[JCL] EPFL Lausanne. Java Constraint Library. `http://liawww.epfl.ch/~akira/JCL/`.

[JG00] Ulrich John and Ulrich Geske. Configuration and reconfiguration of industrial products using conbacon. In *Proc. of 10th International Conference on Computing and Information (ICCI'2000)*, LNCS 1821. Springer, 2000.

[JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages POPL-87*, 1987.

[JMSY92] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(r) system and language. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[JRMS89] D. S. Johnson, C. R.Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation, part i. *Operations Research*, 37:865–892, 1989.

[JRMS91] D. S. Johnson, C. R.Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation, part ii. *Operations Research*, 39:378–406, 1991.

[JSol] ILOG S.A. JSolver `http://www.ilog.de/products/jsolver`.

[KDK85] C. Kincaid, P. Dupont, and A. Kaye. Electronic calendars in the office: An assessment of user needs and current technology. *ACM Transactions on Office Information Systems*, 3(1):89–102, 1985.

[Kow74] Bob Kowalski. Predicate logic as programming language. In *Proc. IFIP Congress 74*, pages 569–574, 1974.

[Mat93] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtula time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[Meea] Meetinmaker. `www.meetingmaker.com`.

[Meeb] MeetingWizard. `www.meetingwizard.org`.

[MSO] Microsoft. MS Outlook. `office.microsoft.com/assistance/offhelp/offxp/outlook`.

[Min] IFComputer. Minerva Internet site. `http://www.ifcomputer.com/MINERVA`.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[Mon96] Eric Monfroy. *Collaboration de solveurs pour la programmation logique à constraints*. PhD thesis, Université Henri Poincaré Nancy, 1996. also availible in english.

[Mon98] Eric Monfroy. The constraint solver collaboration language of bali. In *Proc. FroCoS98*, 1998.

[Mon00] Eric Monfroy. A coordination-based chaotic iteration algorithm for constraint propagation. In *Proc. 15th ACM Symposium on Applied Computing, SAC'00, Track on Coordination Models, Languages and Applications*, 2000.

[Mos90] Peter D. Mosses. Denotational semantics. In van Leeuwen J, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[MOz] The Mozart Consortium. `http://www.mozart-oz.org`.

[MR98] Eric Monfroy and Christophe Ringeissen. Solex: A domain-independant scheme for constraint solver extension. In J. Calmet and J. Plaza, editors, *AISC'98, LNAI 1476*, pages 222–233. Springer-Verlag Berlin Heidelberg, 1998.

[MR99] Eric Monfroy and Jean-Hughes Réty. Chaotic iteration for distributed constraint propagation. In *Proc. 14th ACM Symposium on Applied Computing, SAC'99, Artificial Intelligence and Computational Logic Track*, 1999.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

[MT00] Kurt Mehlhorn and Svben Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Proc. CP2000*, volume LNCS-1894. Springer, 2000.

[Nar01] Alexander Narayek. *Constraint-Based Agents - An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds*. PhD thesis, Technical University of Berlin, Department of Computer Science, 2001. appeared as Springer LNAI 2062.

[NB94] B. Neveu and P. Berlandier. Maintaining arc consistency through constraint retraction, 1994.

[Neu90] U. Neumerkel. Extensible unification by metastructures. Technical report, Institut für Praktische Informatik, Technische Universität Wien, 1990.

[NM93] Klaus Neumann and Martin Morlock. *Operations Research*. Hanser Verlag, 1993.

[Oeb01] Rickard Oeberg. *Mastering RMI*. Wiley, 2001.

[OH83] Ernst-Rüdiger Olderog and C. A. R. Hoar. Specification-oriented semantics for communicating processes. In *Proc. ICALP83*, pages 561–572, 1983.

[PG96] Gilles Pesant and Michel Gendreau. A view of local search in constraint programming. In Springer, editor, *Proceedings CP'96*, number 1118 in LNCS, pages 363–366, 1996.

[pooc] Fraunhofer FIRST. Platfor for object-oriented constraint programming. `http://www.first.fhg.de/~hs/pooc`.

[PS96] Cuno Pfister and Clemens Szyperski. Why objects are not enough. In *Proceedings, International Component Users Conference*, Munich, Germany, 1996. SIGS.

[Pug94] J.-F. Puget. A c++ implementation of clp. Technical report, ILOG, 1994. In Ilog Solver Collected papers.

[Rin01a] Georg Ringwelski. Constraint-based methods in object-oriented programming environments. In *CP'01, Doctoral Programme*, 2001. `http://www.math.unipd.it/~frossi/ringwelski.ps.gz`.

[Rin01b] Georg Ringwelski. Distributed constraint satisfaction with cooperating asynchronous solvers. In *Proc. CP'01*, LNCS 2239, page 777. Springer, 2001.

[Rin01c] Georg Ringwelski. A new execution model for constraint processing in object-oriented software. In *Proc. International Workshop on Functional and (Constraint) Logic Programming (WFLP01)*, 2001.

[Rin02a] Georg Ringwelski. Integrating search objects in asynchronous constraint solving. In *ProcCP'02*, LNCS, page ??? Springer, 2002.

[Rin02b] Georg Ringwelski. Object-oriented constraint programming with j.cp. In Coello Coello, editor, *Advances in Artificial Intelligence, MICAI2002*, LNAI 2313, pages 194–203. Springer, 2002.

[Rin02c] Georg Ringwelski. Posting and retracting consrtaints in asynchronous solvers. In *Second International Workshop on User-Interaction in Constraint Satisfaction, UICS 02*, 2002.

[Rin02d] Georg Ringwelski. Search in asynchronous constraint solving. In *CP'02 Doctoral Programme*, 2002. `http://www.math.unipd.it/~frossi/ringwelski.ps`.

[RM01] Igor Razgan and Amnon Meisels. Distributed forward checking with dynamic ordering. In *Proc. IJCAI01-Workshop on Distributed Constraint Reasoning*, 2001.

[RMI] java.sun.com jGuru. Java RMI Tutorial. `developer.java.sun.com/developer/onlineTraining/rmi/`.

[Rom99] Ed Roman. *Mastering Enterprise JavaBeans*. Wiley, 1999.

[RPD90] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In Luigia Carlucci Aiello, editor, *ECAI'90: Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, Stockholm, 1990. Pitman.

[RS00a] Georg Ringwelski and Hans Schlenker. Type inference in chr programs for the composition of constraint systems. In *15. WLP*, 2000.

[RS00b] Georg Ringwelski and Hans Schlenker. Using typed interfaces to compose chr programs. In *Proc. CP00 Workshop on Rule-Based Constraint Reasoning*, 2000.

[RS02] Georg Ringwelski and Hans Schlenker. Distributed constraint satisfaction with asynchronous solvers. In *ERCIM/CologNet Workshop on Constraint Solving and Constraint Logic Programming*, 2002.

[RWG00] Georg Ringwelski, Armin Wolf, and Ulrich Geske. Ein ansatz fuer low-level finite-domain-constraints in minerva. In *Proceedings of the 14th Workshop Logische Programmierung*, 2000. in german, also published as GMD Report 100.

[Sar93] V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[Sch97] Christian Schulte. Programming constraint inference engines. In *Proc. CP97*, LNCS 1330, pages 519–533. Springer, 1997.

[Sch99] Christian Schulte. Comparing trailing and copying for constraint programming. In Danny de Schreye, editor, *Proc. ICLP99*, pages 275–289. MIT Press, 1999.

[Sch00] Christian Schulte. *Programming Constraint Services.* PhD thesis, Universitt des Saarlandes, Naturwissenschaftlich-Technische Fakultt I, Fachrichtung Informatik, 2000.

[Sch02] Chritian Schulte. *Programming Constraint Services.* LNAI 2302. Springer, 2002. PhD thesis.

[SD98] Sandib Sen and E.H. Durfee. An formal study of distributed meeting scheduling. *Group Decision and Negotiation*, pages 265–289, 1998.

[See01] Frank Seelisch. Heterogeneous constraint problems. In Toby Walsh, editor, *Proc. CP01*, LNCS 2239, page 783. Springer, 2001.

[SICS] Swedish Institute of Computer Science. Sicstus prolog v3, 2001. `http://www.sics.se/sicstus.html`.

[Sim] Fraunhofer FIRST. The SIMONE Project. `http://www.first.gmd.de/plant/2simone.html`.

[Smo95] Gert Smolka. The oz programming model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[Sol00] Christine Solnon. Solving permutation constraint problems with artificial ants. In W. Horn, editor, *Proc. ECAI 2000.* IOS Press, 2000.

[SR00] Hans Schlenker and Georg Ringwelski. New reasons for the introduction of concurrency in prolog. In *Proc. Workshop Concurrency, Specification & Programming 2000 (CS&P2000)*. HU Berlin, 2000.

[SR02] Hans Schlenker and Georg Ringwelski. Pooc - a platform for object-oriemnted constraint programming. In *ERCIM/CologNet Workshop on Constraint Solving and Constraint Logic Programming*, 2002.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog, 2nd Edition*. MIT Press, 1994.

[SSHF00] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922, 2000.

[SSHF01] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Consistency maintainance for abt. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, pages 271–285. Springer LNCS 2239, 2001.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, May 1994. Springer-Verlag.

[Ste99] Kathleen Steinhöfel. *Stochastic Algorithms in Scheduling Theory*. PhD thesis, TU Berlin, 1999.

[TYC00] Ka Boon Ng Tee Yong Chew, Martin Henz. A toolkit for constraint-based inference engines. In S.C. Vitor E. Pontelli, editor, *Proccedings of the Second International Workshop on Practical Aspects of Declarative Languages (PADL '00)*, volume 1753 of *Lecture Notes in Computer Science*, pages 185–199, Boston, MA, USA, January 17-18 2000. Springer.

[Voß00] Stefan Voß. Meta-heuristics: The state of the art. In Alexandder Narayek, editor, *Local Search in Planning and Scheduling*, LNAI 2148, pages 1–23. Springer, 2000. Invited Paper.

[VS94] Gerard Verfaillie and Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *National Conference on Artificial Intelligence*, pages 307–312, 1994.

[Wal75] D.L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. In P.H. Winston, editor, *The Psychlogy of Computer Vision*. McGraw Hill, 1975.

[Wal96] Richard J. Wallace. Analysis of heuristic methods for partial constraint satisfaction problems. In *Principles and Practice of Constraint Programming*, pages 482–496, 1996.

[Wal97] Toby Walsh. Depth-bounded discrepancy search. In *Proc. IJCAI*, pages 1388–1395, 1997.

[Wal00] Toby Walsh. Sat v csp. In *Proc. CP2000*, volume LNCS-1894, pages 441–456. Springer, 2000.

[WDH95] Matthew L. Ginsberg William D. Harvey. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, pages 607–615, Montréal, Québec, Canada, August 20-25 1995. Morgan Kaufmann, 1995.

[WF98] Richard J. Wallace and Eugene C. Freuder. Stable solutions for dynamic constraint satisfaction problems. *Lecture Notes in Computer Science* 1520, p.447ff, 1998.

[WGG00] Armin Wolf, Thomas Gruenhagen, and Ulrich Geske. On incremental adaption of constraint handling rule derivation after constraint deletions. *Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, 2000.

[WNS97] M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, 1997.

[Wol99] Armin Wolf. *Adaptive Constraintverarbeitung mit Constraint-Handling-Rules.* PhD thesis, TU Berlin, 1999.

[Wol01] Armin Wolf. Adaptive constraint handling with chr in java. In Toby Walsh, editor, *Proc. CP2001*, LNCS 2239, pages 256–270. Springer, 2001.

[YD98] Makoto Yokoo and Edmund H. Durfee. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5), 1998.

[YH00] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

[Zoe02] Peter Zoeteweij. A coordination-based framework for parallel constraint solving. In *ERCIM CologNet Workshop on Constraint Solving and Constraint Logic Programming*, 2002.