INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

# Ein Überblick über die Constraint-Programmierung

Ozan Ilgin Tanriverdi

**Bachelorarbeit**

**Betreuer** Prof. Dr. François Bry
**Mentor** Thomas Prokosch

**Abgabe am** 01.08.2022

# Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 01.08.2022 Ozan Ilgin Tanriverdi

# Abstract

Constraint Programming (CP) can be described as an extension of logic programming and combines logic programming with constraint solving. Since its proposal as „CLP(X) Scheme“ by Joxan Jaffar and Jean-Louis Lassez in 1987, CP became one of the focus points for research in logic programming. One of the reasons for attention was the wide range of application areas in which CP proved to be useful. The application areas of CP ranges from operation research to theorem proving and even to bioinformatics. Two of the reasons for this wide spectrum of use cases are declarativity and the high expressive power CP offers. First, declarativity of CP allows programmers to take advantage of the typically short development time common in logic programming. Second, a higher expressive power compared to logic programming is achieved by allowing multiple variable domains.

In this thesis, different aspects of this programming paradigm will be introduced. Chapter 1 describes the goal of this thesis and gives an overview of related work. Chapter 2 contains an introduction to logic programming and deals with the functionality of CP in four subchapters. Two core aspects of CP, constraint propagation and the search which then follows, are introduced in this chapter. Chapter 3 introduces some interesting extensions of CP, and chapter 4 presents a variety of CP languages. In the last chapter, some use cases of CP from different application areas are explored more in detail.

# Zusammenfassung

Constraint-Programmierung (CP) kann als eine Erweiterung der Logikprogrammierung gesehen werden, die dieses deklarative Paradigma mit Techniken zum Lösen von Constraints kombiniert. Seit ihrer Einführung als CLP(X)-Schema von Joxan Jaffar und Jean-Louis Lassez im Jahr 1987 wurde CP zu einem der Schwerpunkte für die Forschung in der Gemeinschaft der Logikprogrammierung. Einer der Gründe für dieses Interesse war das breite Spektrum an Anwendungsbereichen, in denen sich CP als nützlich erwiesen hat. Die Anwendungsgebiete von CP reichen von Optimierungsrechnung über Satzbeweise bis hin zur Bioinformatik. Zwei der Gründe für das breite Spektrum an Anwendungsfällen sind die Deklarativität und die hohe Ausdruckskraft, die CP den Programmierern anbietet. Die Deklarativität von CP ermöglicht es den Programmierer, die typischerweise kurze Entwicklungszeit von Logikprogrammierung ebenso auszunutzen. Weiters bietet CP eine höhere Ausdruckskraft im Vergleich zu Logikprogrammierung, indem eine Vielzahl von Variablendomänen zugelassen und die Benutzer nicht auf eine Domäne beschränkt werden.

In dieser Arbeit werden verschiedene Aspekte dieses Programmierparadigmas vorgestellt. Kapitel 1 beschreibt das Ziel dieser Arbeit und gibt einen Überblick über verwandte Arbeiten. Kapitel 2 enthält eine Einführung in die Logikprogrammierung und befasst sich in vier Unterkapiteln mit der Funktionalität von CP. Zwei Kernaspekte von CP, die Constraint Propagation und die darauffolgende Suche, werden in diesem Kapitel vorgestellt. In Kapitel 3 werden einige wichtige Erweiterungen von CP vorgestellt, und in Kapitel 4 wird eine Vielzahl von CP-Sprachen präsentiert. Im letzten Kapitel werden einige Anwendungsfälle von CP aus verschiedenen Anwendungsbereichen näher beleuchtet.

# Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während meines ganzen Bachelorstudiums und besonders bei der Verfassung dieser Bachelorarbeit unterstützt haben.

Zuerst möchte ich mich bei Herrn Prof. Dr. François Bry bedanken, der mir die Möglichkeit gegeben hat, zu diesem spannenden Thema meine Bachelorarbeit zu schreiben.

Als nächstes möchte ich mich bei meinem Betreuer Dipl.-Ing. Thomas Prokosch bedanken, der mich von der Themensuche bis zur Abgabe der Arbeit unterstützt und diesen einschüchternden Prozess für mich erleichtert hat.

Schließlich gilt mein Dank meiner Familie, die für mich immer da war und mich nicht nur während meines Studiums, sondern während meines ganzen Lebens unterstützt hat.

# Inhaltsverzeichnis


**1 Einleitung** **1**
- 1.1 Ziel der Arbeit . . . 1
- 1.2 Verwandte Arbeiten . . . 1

**2 Konzepte der Constraint-Programmierung** **5**
- 2.1 Allgemeine Konzepte der Logikprogrammierung . . . 5
- 2.2 Constraints . . . 10
  - 2.2.1 Repräsentation und Semantik von Constraints . . . 10
  - 2.2.2 Typische Domänen der Constraint-Programmierung . . . 13
  - 2.2.3 Constraint Propagierung . . . 15
  - 2.2.4 Auflösen von Constraints . . . 18

**3 Fortgeschrittene Techniken** **21**
- 3.1 Inkrementelle Constraint-Programmierung . . . 21
- 3.2 Nebenläufige Constraint-Programmierung . . . 24

**4 Sprachen für Constraint-Programmierung** **29**
- 4.1 CHIP . . . 29
- 4.2 CLP(R) . . . 30
- 4.3 Constraint Handling Rules (CHR) . . . 31

**5 Beispielhafte Verwendungen** **33**
- 5.1 Automatisierte Planung . . . 33
- 5.2 Verifikation Digitaler Schaltungen . . . 34

**6 Fazit** **37**
- 6.1 Zusammenfassung . . . 37
- 6.2 Künftige Forschungsthemen . . . 37

**Literaturverzeichnis** **39**

KAPITEL 1

# Einleitung

## 1.1 Ziel der Arbeit

Constraint-Programmierung (CP) liegt im Schnittpunkt von Logikprogrammierung und Constraintlösen (engl. constraint solving). Das Lösen von Constraints ist vermutlich nicht geläufig, aber auch die deklarative Art der Programmierung, die in Constraint-Programmierung zum Einsatz kommt, muss näher erläutert werden. Doch auch ohne diese Erläuterungen ist vermutlich bereits jetzt die Neugierde an Constraint-Programmierung geweckt. Auf den ersten Blick sieht CP wie eine „indirekte“ Art der Programmierung aus, da im Programm angegeben wird, welche Art von Beschränkungen Variablen erfüllen sollen, aber nicht, wie das Problem gelöst werden soll. Constraint-Programmierung kann in vielen Problemen unserer Zeit Anwendung finden. Außerdem ist Constraint-Programmierung eng mit vielen anderen Fachgebieten stark verknüpft.

Ziel dieser Arbeit ist es, einen Überblick über dieses Programmierparadigma zu geben. Der Stil der Abfassung ist so gewählt, dass sich der Inhalt an jeden Leser und jede Leserin mit Grundlagenkenntnissen der Informatik richtet. Dafür wird zuerst der theoretische Hintergrund von CP zusammen mit ihrer Notwendigkeit und geschichtlicher Entwicklung erläutert. Danach werden einige Erweiterungen und zahlreiche Sprachen für Constraint-Programmierung vorgestellt. Schließlich wird anhand von unterschiedlichen Beispielen gezeigt, dass CP in einer Vielzahl von Anwendungsgebieten Vorteile bringen kann.

## 1.2 Verwandte Arbeiten

Das Konzept der Constraints war schon vor der Einführung der Constraint-Programmierung existent in der Informatik. Die Ursprünge der Constraints lassen sich bis in die späten 1970er Jahre zurückfolgen. An Manipulation und Propagierung von Constraints wurde bereits im Rahmen der künstlichen Intelligenz geforscht, um intelligentere Suchverfahren zu ermöglichen [28, 69].

Zwei wichtige Produkte dieser Forschung waren Sketchpad [71] und Thinglab [12]. Sketch-Pad wurde von Ivan Edward Sutherland im Jahr 1963 als Teil seiner Doktorarbeit entwickelt und war vermutlich die früheste Bemühung, die als eine Constraint-Sprache klassifiziert werden kann. Sketchpad war ein interaktives System für das Zeichnen von geome-

trischen Objekten und gilt als ein Pionier für das Fachgebiet Mensch-Maschine-Interaktion. In diesem System wurden Constraints verwendet, um die geometrischen Eigenschaften der Objekte zu beschränken.

Ein nachfolgendes Projekt, ThingLab, war ein grafisches Simulationslab, wo Constraints verwendet wurden, um die Beziehungen zu spezifizieren, die zwischen den Teilen einer Simulation gelten müssen [11]. Das Wesentliche von ThingLab war die folgende Neuigkeit: Bei der Manipulation eines grafischen Objekts wird ein Plan erstellt, um die entsprechenden Constraints für den geänderten Teil des Objekts schnell neu zu lösen. Dieser Plan wurde dann wiederholt ausgeführt, während die Manipulation fortgesetzt wurde [43]. Dies stellt eine einfache Form der Inkrementalität dar, eine Eigenschaft, die in mehreren CP-Systemen verwendet wird (siehe Kapitel „3.1 Inkrementelle Constraint-Programmierung“).

Außer dieser grafischen Anwendung lassen sich viele Alltagsprobleme wie Planung, Zuteilungsprobleme, Fehlerdiagnose oder Hardware-Design mit Constraints so formulieren, dass eine Lösung aller Constraints die Problemstellung erfüllen muss [30]. Ein oft verwendeter Ansatz, um ein Constraintproblem zu lösen, war die Formulierung eines spezifischen Programms mit einer prozeduralen Sprache. Dies erfordert aber großen Aufwand bei der Programmentwicklung und später bei Wartung, Modifikation oder Erweiterung des Programms. Dies war eine Motivation für die Entwicklung von Constraint-Programmierung [28].

Mit der Arbeit von Joxan Jaffar und Jean-Louis Lassez aus dem Jahr 1987 [42] wurde diese Motivation formalisiert und das Schema der Constraint-Programmierung als CLP(X) eingeführt. Dieses Schema besagt, dass Unifikation, der Hauptmechanismus der Logikprogrammierung, durch Constraintbehandlung (engl. constraint handling) ergänzt werden soll. Außerdem wurde in diesem Artikel gezeigt, wie die Semantik der Logikprogrammierung erweitert werden muss, damit unterschiedliche Datentypen wie Boolean oder die Menge der ganzen Zahlen in die Logikprogrammierung eingeführt werden können [28]. In der Bezeichnung CLP(X) ist X tatsächlich als eine Variable zu sehen, die mit verschiedenen Datentypen initialisiert werden kann (wie etwa CLP(FD) für ganze Zahlen oder CLP(B) für Boolean usw.).

Obwohl es schwer nachzuvollziehen ist, welche Sprache die erste CP-Sprache ist, wird oft Prolog II als die erste CP-Sprache angenommen. Prolog II wurde in den frühen 1980er Jahren von Alain Colmerauer entwickelt [15]. Als der Nachfolger von Prolog kann Prolog II zusätzlich zu den Termgleichheiten auch bei Ungleichheiten der Termen auf Erfüllbarkeit entscheiden. Weil diese Entscheidung abseits der Möglichkeiten von Unifikation ist, kann man Ungleichheiten als eine „primitive“ Art von Constraints ansehen [32].

Eine weitere Sprache ist CHIP, entwickelt in European Computer Industry Research Centre (ECRC) [20]. Eine der Besonderheiten von CHIP ist, dass es die erste Sprache ist, bei der Constraint Propagierung umgesetzt wurde [37]. Constraint Propagierung wird in Kapitel 2.2.3 ausführlicher behandelt, aber für den Moment kann man sich diese als Sammlung von Methoden vorstellen, die zur Einschränkung von Variablendomänen dienen. In CHIP wurden zusätzlich arithmetische Constraints eingeführt und auf endlichen Domänen angewendet [28]. Viele Probleme aus dem Fachgebiet der Optimierungsrechnung [19, 18] konnten mit Hilfe der Sprache CHIP gelöst werden. Es existieren auch Beispielprobleme aus den Fachgebieten wie Schaltungsentwurf [67] und Finanz [49], für die CHIP sich als nützlich erwiesen hat. Dieser Erfolg von CHIP hat dazu geführt, dass CP mehr Interesse in industriellen Anwendungen bekam.

Das nächste wesentliche CP-System ist CLP(R). Die Sprache wurde an der Monash Universität, von IBM Yorktown Heights und der Carnegie Mellon Universität entwickelt [44]. Die Domäne von CLP(R) ist die Menge der reellen Zahlen [28]. Vorrangiges Ziel der Implementierung der Sprache war es, das praktische Potenzial des CLP(X)-Schemas unter Beweis zu stellen [30].

Prolog III, der Nachfolger von Prolog II und damit ebenso ein CP-System, wurde an der

Universität Aix-Marseille und an Prologia in Frankreich entwickelt [16]. Prolog III erlaubt drei Constraintdomänen: Rationale Zahlen, Boolean und Zeichenketten. Als weitere CP-Sprachen könnte man Trilogy [76], CAL [2] und BNR-Prolog [9] nennen.

Constraint-Programmierung kann auch inkrementell und nebenläufig verwendet werden, worauf in Kapitel 3 näher eingegangen wird. Zwei weitere interessante Themen sind die stochastische Constraint-Programmierung [79], die aus der Kombination von Constrainterfüllbarkeit und stochastischer ganzzahliger linearer Optimierung zustande kam, und Constraint-Funktionale-Programmierung [52], die funktionale Programmierung mit Constraint-Programmierung kombiniert.

Wichtige Überblicksartikel und Bücher zu Constraint-Programmierung müssen auch genannt werden. Die Arbeit von Joxan Jaffar und Jean-Louis Lassez aus dem Jahr 1987 [42] und „Constraint logic programming languages“ von Jacques Cohen [14] sind wichtige Quellen aus den frühen Jahren der Constraint-Programmierung. Obwohl man die letztgenannte Abfassung als einen Überblicksartikel wahrnehmen kann, liegt der Fokus auf Syntax und Semantik von CP-Sprachen. Das damals neu veröffentlichte Schema von CP-Sprachen wird ausführlich diskutiert. Auf die Techniken des Constraintlösens und Suche wird wenig eingegangen. Syntax und Semantik von CP-Sprachen werden anhand von zahlreichen Beispielen verständlich erklärt.

Als weiteren ausführlichen Überblicksartikel kann man „Constraint Logic Programming - An Informal Introduction“ von Thomas Frühwirth et al. [28] nennen. Der Inhalt ist sehr umfangreich und führt dazu, dass man ein gutes Verständnis über das Thema Constraint-Programmierung bekommt. Die Beispiele sind gut gewählt, wie etwa die „Voll-Addierer-Schaltung“ (engl. full adder circuit), die das Anwendungsgebiet der Boolean-Domäne verständlich erklärt.

Der umfangreichste Überblicksartikel über Constraint-Programmierung ist wahrscheinlich „Constraint logic programming: a survey“ von Joxan Jaffar und Michael J. Maher [43]. Dieser umfangreiche Artikel deckt viele Aspekte von CP ab. Der Inhalt ist in drei Teile unterteilt: Theorie, Umsetzungsprobleme und Programmierung für Anwendungen. Zum Verständnis dieses Artikels sind jedoch Vorkenntnisse in Logikprogrammierung definitiv notwendig.

Ein weiterer, interessanter Artikel ist „Logic, Optimization and Constraint Programming“ von John N. Hooker [41]. Der Artikel stellt den Vergleich zwischen Optimierung und Constraint-Programmierung dar, insbesondere im Hinblick darauf, wie logische Schlussfolgerungen in diesen Disziplinen verwendet werden. Außerdem sind „Constraint satisfaction using constraint logic programming“ von Pascal Van Hentenryck [39], „Constraint (Logic) Programming: A Survey on Research and Applications“ von Francesca Rossi [63], „Constraint Logic Programming“ von Marco Gavanelli und Francesca Rossi [32], „Constraint Logic Programming“ von Helmut Simonis [68] Überblicksartikel, die sehr empfehlenswert sind.

Zum Schluss seien noch einige Bücher genannt, die sich mit CP beschäftigen: „Principles of Constraint Programming“ [3], „Handbook of Constraint Programming“ [62] und „Constraint logic programming using ECLiPSe“ [4]. Das „Handbook of Constraint Programming“ ist sehr umfangreich und enthält zahlreiche Anwendungsbeispiele aus der Industrie. Die beiden erstgenannten Bücher sind zum Einstieg in das Thema empfehlenswert. Das dritte Buch „Constraint logic programming using ECLiPSe“ führt auch in Logikprogrammierung und in Prolog ein. Weiters dient es als Handbuch für die Sprache ECLiPSe.

Weitere Buchempfehlungen sind in etwa: „Essentials of Constraint Programming“ [27], „Constraint Satisfaction in Logic Programming“ [37], „Programming with Constraints - An Introduction“ [54].

KAPITEL 2

# Konzepte der Constraint-Programmierung

## 2.1 Allgemeine Konzepte der Logikprogrammierung

Constraint-Programmierung liegt im Schnittpunkt mehrerer Fachgebiete wie etwa Logikprogrammierung, Constraintlösen (engl. constraint solving) und Optimierungsrechnung (engl. operations research). Jedoch ist es die Logikprogrammierung, von der die wichtigsten Konzepte in Constraint-Programmierung wie Syntax und Semantik inspiriert wurden. In den folgenden einführenden Beispielen wird daher Prolog zur Erläuterung von Konzepten verwendet.

Logikprogrammierung wurde in den frühen 1970er Jahren als eine Folge der früheren Arbeiten im Bereich des automatischen Theorembeweisens und der künstlichen Intelligenz eingeführt [17]. Prolog wird oft als die erste Logikprogrammierungssprache genannt [68] und wurde im Jahr 1972 von Alain Colmerauer und Robert Kowalski entwickelt [60]. Ursprünglich war das Ziel der Entwicklung keine Programmiersprache, sondern ein Produkt zur Verarbeitung natürlicher Sprachen (nämlich Französisch), welches schließlich zu einer Mehrzwecksprache namens Prolog weiterentwickelt wurde [17].

Um Logikprogrammierung verstehen zu können, müssen die Unterschiede zwischen den Begriffen Berechnung (engl. computation) und Schließen (engl. deduction) untersucht werden. Bei einer Berechnung ist zu Beginn ein Ausdruck gegeben und anhand einer festen Menge von Regeln wird ein Ergebnis generiert [60]. Zum Beispiel, $24 * 17 \rightarrow (24 * 10) + (24 * 7) \rightarrow 240 + 168 \rightarrow 408$ zeigt eine mögliche Vorgehensweise für die Multiplikation zweier natürlicher Zahlen, wo das Ergebnis unter Benutzung fester Regeln generiert wird. Beim Schließen jedoch wird mit einer Vermutung angefangen und mit Hilfe einer festen Menge von Regeln (Axiome und Schlussregeln) versucht, einen Beweis für diese Vermutung zu konstruieren [60]. Das heißt, Berechnung kann als ein mechanischer und Schließen als ein kreativer Prozess wahrgenommen werden. In diesem Zusammenhang ist die Logikprogrammierung als eine Kombination von diesen Konzepten zu sehen [60].

Trotz dieser Beobachtung kann Schließen als eine Art Berechnung angesehen werden. Dies ist dann der Fall, falls eine Strategie für die Beweissuche fixiert wird, sodass vom kreativen Aspekt des Schließens Abstand genommen wird. Diese Sichtweise bildet die Grundlage der Logikprogrammierung. Die Berechnung von logischen Programmen erfolgt durch Beweissuche nach einer festen Strategie [60]. Diese Algorithmen können implementiert und als Beweissuche ausgeführt werden.

Eine der wichtigsten Ideen der Logikprogrammierung ist, dass ein Algorithmus zwei disjunkte Komponenten besitzt, nämlich Logik und Steuerung. Mit Logik wird das zu lösende Problem ausgedrückt, und die Steuerung besagt, wie dieses Problem gelöst werden soll. Das Endziel der Logikprogrammierung ist, dass der Programmierer nur die logische Komponente eines Logikprogramms angibt und die Steuerung vom Interpreter des Logikprogramms übernommen wird [51].

Drei grundlegende Eigenschaften der Logikprogrammierung sind für die Constraint-Programmierung von größter Relevanz [68]. Die erste Eigenschaft ist die sogenannte relationale Form der Logikprogrammierung, die es ermöglicht, Prädikate zu definieren, ohne dass die Ein- und Ausgangsargumente von vornherein festgelegt werden. Die zweite Eigenschaft ist Nicht-Determinismus, der ein Baum-Suchverfahren (mit Backtracking) als Teil des Berechnungsmechanismus ermöglicht. Zu guter Letzt ist der Algorithmus der Unifikation, die zur Lösung von Gleichungen mit (uninterpretierten) symbolischen Termen verwendet wird, fester Bestandteil der Logikprogrammierung.

Zur Darstellung der logischen Zusammenhänge eines zu lösenden Problems wird in Prolog die Sprache der Prädikatenlogik verwendet. Die Basis für das Wissen besteht aus logischen Formeln (sogenannten Klauseln). Es gibt zwei Arten von Klauseln in Programmen: Fakten, die Aussagen über Objekte treffen, und Regeln, die einem erlauben, aus bekannten Fakten auf neue Fakten zu schließen [33]. Zum Beispiel könnte der folgende Ausdruck

```
mutterVon(elisa, jana).
```

einen Fakt darstellen, der bedeutet, dass Elisa die Mutter von Jana ist. Ein Fakt besteht immer aus dem Namen der Eigenschaft oder Relation (dem sogenannten Prädikatensymbol) gefolgt von Objekten in einem Klammerpaar, die diese Eigenschaft haben. Danach folgt ein Punkt.

Zur Ausführung eines logischen Programms stellt der Anwender Anfragen an das Programm [33]. Dies kann wie eine „dialogorientierte“ Programmiersprache wahrgenommen werden. Im Kontext der Verwandtschaftsrelationen, wie im letzten Beispiel, wäre eine mögliche Frage: „Ist Hans der Vater von Jakob?“ In Prolog beginnen Anfragen mit dem zusammengesetzten Symbol „?−“. Danach folgen Aussagen (Formeln), die bewiesen werden sollen. Zum Schluss der Anfrage folgt ein Punkt. In Prolog könnte die eben beschriebene Anfrage folgendermaßen aussehen:

```
?- vaterVon(hans, jakob).
```

Die Antwort auf diese Anfrage liefert einen logischen Beweis anhand der vom Programmierer gegebenen Problembeschreibung und der Regeln der Programmiersprache, wobei die Techniken „Unifikation“ und „Resolution“ zum Einsatz kommen [33]. Die Antwort zu obiger Anfrage kann *wahr* (engl. true) oder $falsch$ (engl. false) ergeben.

In der Logikprogrammierung können auch Variablen verwendet werden, um anspruchsvollere Anfragen zu formulieren oder Fakten zu erstellen, die für alle Objekte der Wissensbasis gelten müssen. In Prolog beginnen Variablen mit Großbuchstaben oder einem Unterstrich [33]. Ein Fakt der Form

```
mensch(X).
```

würde bedeuten, dass diese Eigenschaft für jede mögliche Belegung der Variable $X$ gilt.

Variablen ermöglichen die Formulierung anspruchsvoller Anfragen. Zum Beispiel kann die Anfrage für die Frage „Wer ist der Vater von Jakob?“ mit Variablen gebildet werden und könnte wie folgt aussehen:

```
?- vaterVon(X, jakob).
```

Die Antwort zu dieser Anfrage würde der Name von Jakobs Vater als Variablenbindung *X* sein.

Als weiteres Beispiel dient eine zweistellige Relation mit zwei Variablen:

```
?- mutterVon(X, Y).
```

Diese Anfrage liefert alle Paare $(X,Y)$ von Menschen, die in einer Mutter-Kind-Relation sind.

Neben Fakten gibt es auch Regeln. Regeln dienen dazu, aus bekanntem Wissen neues Wissen herzuleiten [33]. Regeln bestehen aus zwei Teilen, die durch das zusammengesetzte Symbol „: −“ getrennt sind. Der linke Teil heißt Kopf der Regel und der rechte Teil heißt Rumpf der Regel [33]. Der Kopf stellt eine Schlussfolgerung dar und der Rumpf enthält die Voraussetzungen für diese Schlussfolgerung. Wenn mehr als eine Voraussetzung existiert, so werden diese mit Kommas getrennt. Diese Kommas bezeichnen eine Konjunktion der Voraussetzungen. Das heißt, alle Voraussetzungen müssen zutreffen.

Als Beispiel für eine Regel kann eine Kind-Relation aus der vorherigen imaginären Gruppe von Personen untersucht werden. Natürlich wäre es möglich, diese Relation als Folge von Fakten wie bei der Mutter-Kind-Relation zu definieren, aber es ist kürzer und übersichtlicher, eine Regel dafür zu verwenden. Bei unendlich vielen Objekten oder dynamischen Objektmengen kommt ohnehin nur eine Regel anstatt einer Aufzählung der Fakten in Frage.

Sei eine Kind-Relation folgendermaßen definiert:

```
kindVon(X, Y) :- mutterVon(Y, X).
kindVon(X, Y) :- vaterVon(Y, X).
```

In diesem Beispiel wird eine Relation mit Hilfe von zwei Regeln definiert. Aus der Tatsache, dass eine Person Y die Mutter oder der Vater von einer Person X ist, kann gefolgert werden, dass Person X das Kind von Person Y ist. Das „oder“ in diesem Szenario lässt sich in der Logikprogrammierung so formulieren, dass für eine Schlussfolgerung mehr als eine Regel (mit dem gleichen Kopf) definiert wird.

Mit Hilfe der Kind-Relation kann eine Enkelkind-Relation definiert werden. Die folgende Regel besagt: „Eine Person A ist Enkelkind eines Großelternteils C, falls die Person A das Kind der Person B, wobei die Person B auch das Kind von Person C ist.“

```
enkelkindVon(A, C) :- kindVon(A, B), kindVon(B, C).
```

Dieses Bespiel veranschaulicht, wie eine Regel mit mehr als einer Voraussetzung aussehen könnte.

In Prolog werden Regeln beim Beweisen rückwärts angewendet [33]. Um zu zeigen, dass die linke Seite einer Regel gilt, muss gezeigt werden, dass die rechte Seite der Regel gilt (engl. backward chaining). Sei die folgende Anfrage gegeben:

```
?- enkelkindVon(Finn, Franz).
```

Um diese Aussage zu beweisen, muss man aufgrund der Regel für *enkelkindVon* ein X finden, so dass die Aussagen

```
kindVon(Finn, X), kindVon(X, Franz).
```

wahr sind.

## Grundlagen der Prädikatenlogik

Die Logikprogrammierung basiert auf den Konzepten der Prädikatenlogik erster Stufe. Im Folgenden werden insbesondere jene Aspekte der Prädikatenlogik erster Stufe beschrieben, die für die Constraint-Programmierung von besonderer Bedeutung sind. Dabei wird zwischen Syntax und Semantik der Prädikatenlogik unterschieden.

## Syntax der Prädikatenlogik

**Logische Symbole.** Das Alphabet der Prädikatenlogik besteht aus den folgenden Klassen von Symbolen [50]: Junktoren, Quantoren, technische Zeichen, Variablensymbole, Konstantensymbole, Funktionssymbole, Prädikatensymbole. Junktoren sind Negation $\neg$, Konjunktion $\wedge$, Disjunktion $\vee$, Implikation $\Rightarrow$ und Äquivalenz $\Leftrightarrow$. Quantoren sind der Allquantor $\forall$ und der Existenzquantor $\exists$. Letztlich sind die folgenden Symbole als technische Zeichen verfügbar: „(“, „)“ und „,“. Die üblichen Notationen für die restlichen Symbole sind folgendermaßen [50]:

- Variablensymbole werden mit Großbuchstaben wie $X, Y, Z$ bezeichnet, die normalerweise aus dem späteren Teil des (lateinischen) Alphabets stammen.
- Konstantensymbole werden mit Kleinbuchstaben wie $a, b, c$ bezeichnet, die normalerweise aus dem früheren Teil des (lateinischen) Alphabets stammen.
- Funktionssymbole werden mit Kleinbuchstaben wie $f, g, h$ bezeichnet.
- Prädikatensymbole werden mit Kleinbuchstaben wie $p, q, r$ bezeichnet.

Sei zudem die Menge der Variablensymbole abzählbar.

**Terme.** Die Menge der Terme (über einem gegebenen Alphabet) wird induktiv folgendermaßen definiert [13]:

- Eine Variable ist ein Term.
- Eine Konstante ist ein Term.
- Seien $f$ ein Funktionssymbol mit Stelligkeit $n > 0$ und $t_1, .., t_n$ Terme, dann ist $f(t_1, .., t_n)$ auch ein Term.

**Atome.** Seien $n \geq 0$, $p$ ein n-stelliges Prädikatensymbol und $t_1, .., t_n$ Terme. Dann ist $p(t_1, .., t_n)$ ein Atom. Für $n = 0$ kann das Atom $p()$ oder $p$ geschrieben werden und wird elementares Atom genannt [13].

**Formel.** Eine (wohlgeformte, engl. well-formed) Formel (über einem gegebenen Alphabet) wird induktiv wie folgt definiert [51]:

- Ein Atom ist eine Formel.
- Falls F und G Formeln sind, so auch $\neg F, F \wedge G, F \vee G, F \Rightarrow G, F \Leftrightarrow G$.
- Falls X eine Variable und F eine Formel ist, dann sind $\forall XF$ und $\exists XF$ Formeln. In diesem Fall ist die Variable X nicht frei in F.

**Geschlossene Formel.** Eine geschlossene Formel ist eine Formel, in der keine Variable frei vorkommt [51].

Die Menge aller Formeln, die unter Verwendung eines gegebenen Alphabets A konstruiert werden können, heißt Sprache erster Ordnung, die durch A gegeben ist [51].

**Literal.** Ein Literal ist ein Atom oder die Negation eines Atoms [51].

**Klausel.** Seien $m$ und $s$ natürliche Zahlen mit $m, s \geq 0$. Eine Klausel ist eine Formel der Form $\forall x_1 ... \forall x_s (L_1 \vee ... \vee L_m)$, wo jedes $L_i$ ein Literal und $x_1, ..., x_s$ Variablen, die in $L_1 \vee ... \vee L_m$ vorkommen, sind [51].

Klauseln treten in der Logikprogrammierung häufig auf, daher gibt es für diese eine weitere, kürzere Notation: Die Klausel $\forall x_1 ... \forall x_s (A_1 \vee ... \vee A_k \vee \neg B_1 \vee ... \vee \neg B_n)$ mit den Atomen $A_1, ...A_k, B_1, ..., B_n$ und Variablen $x_1, ..., x_s$, die in den Atomen vorkommen, wird als $A_1, ..., A_k \leftarrow B_1, ..., B_n$ geschrieben [51]. Eine Formel in dieser Notation nennt man eine Regel [13]. Die linke Seite der Formel mit den Atomen $A_1, ..., A_k$ nennt man Klauselkopf, die rechte Seite der Formel mit den Atomen $B_1, ..., B_n$ nennt man Klauselrumpf. Dabei entsprechen die Kommas im Klauselkopf einer Disjunktion ($\vee$) und die Kommas im Klauselrumpf einer Konjunktion ($\wedge$).

**Programmklausel.** Seien $A, B_1, ..., B_n$ Atome. Eine Programmklausel ist eine Klausel der Form $A \leftarrow B_1, ..., B_n$ [51]. $A$ wird der Kopf und $B_1, ..., B_n$ wird der Rumpf der Programmklausel genannt. Eine Programmklausel mit einem leeren Rumpf, $A \leftarrow$, nennt man eine Einheitsklausel.

Anhand der definierten Begriffen können Logik-Programme definiert werden: Ein Logik-Programm ist eine endliche Menge von Programmklauseln [51]. In einem Logik-Programm heißen die Sammlung aller Programmklauseln mit demselben Prädikat $p$ im Kopf die Definition von $p$.

**Zielklausel.** Sei $n$ eine natürliche Zahl mit $1 \leq i \leq n$. Eine Zielklausel ist eine Klausel der Form $\leftarrow B_1, ..., B_n$, das heißt, eine Klausel mit leerem Kopf [51]. Jede $B_i$ heißt ein Teilziel (engl. subgoal) von der Zielklausel.

## Modelle von Logik-Programmen

Um über den Wahrheitsgehalt von Formeln diskutieren zu können, ist es zunächst notwendig, jedem der Symbole in der Formel eine Bedeutung zuzuordnen. Einige Strukturen wie Quantoren oder Junktoren besitzen feste Bedeutungen, aber die Bedeutungen der Konstanten, Funktionen und Prädikate können variieren. Eine Interpretation kann als eine Belegung dieser Strukturen mit Werten aus der entsprechenden Domäne wahrgenommen werden. Besonders interessant sind diejenigen Interpretationen, für die die Formel eine wahre Aussage ausdrückt. Eine solche Interpretation nennt man ein Modell der Formel. Formaler definiert:

**Interpretation.** Eine Interpretation einer Sprache $L$ erster Stufe ist ein 3-Tupel der Form (D,I,V) bestehend aus [13]:

- Einer nicht-leeren Menge $D$, Domäne der Interpretation genannt.
- Einer Funktion I, die die Funktionssymbole und Prädikatensymbole auf die Domäne D abbildet.
- Einer Variablenzuweisung V nach D.

**Modell.** Sei $I$ eine Interpretation einer Sprache erster Stufe $L$ und sei $F$ eine geschlossene Formel von $L$. Dann ist $I$ ein Modell für $F$, falls der Wahrheitswert von $F$ in Bezug auf $I$ wahr ist [51].

**Grundterm.** Ein Grundterm (engl. ground term) ist ein Term, der keine Variablen enthält.

**Herbrand-Universum.** Sei $L$ eine Sprache erster Stufe mit mindestens einer Konstante. Das Herbrand-Universum $U_L$ für $L$ ist die Menge aller Grundterme, die durch Konstanten und Funktionen in $L$ geformt werden können [51].

Es sei ein Beispiel gegeben. Eine Sprache erster Stufe $L$ habe die einstelligen Prädikate $p$ und $r$, das zweistellige Prädikat $q$, die Funktionen $f, g$ und die Konstante $a$. Das Herbrand-Universum von $L$ ist die folgende Menge:

$$\{a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), ...\}$$

**Grundatom.** Ein Grundatom (engl. ground atom) ist ein Atom, welches keine Variablen enthält.

**Herbrand-Basis.** Sei $L$ eine Sprache erster Stufe. Die Herbrand-Basis (engl. Herbrand base) $B_L$ für $L$ ist die Menge aller Grundatome, die mit Termen von L geformt wird [51].

Für das vorherige Beispiel würde die Herbrand-Basis folgendermaßen aussehen:

$$\{p(a), q(a,a), r(a), p(f(a)), p(g(a)), q(a, f(a)), q(f(a), a), ...\}$$

**Herbrand-Interpretation.** Sei $L$ eine Sprache erster Stufe. Eine Interpretation für L ist eine Herbrand-Interpretation, wenn die folgenden Bedingungen erfüllt sind [51]:

- Die Domäne der Interpretation ist das Herbrand-Universum $U_L$.
- Konstanten in $L$ werden sich selbst zugewiesen.
- Falls $f$ eine $n$-stellige Funktion in $L$ ist, dann wird $f$ der Abbildung von $(U_L)^n$ auf $U_L$ mit dem Vorschrift $(t_1, ..., t_n) \rightarrow f(t_1, ..., t_n)$ zugewiesen.

**Herbrand-Modell.** Sei $L$ eine Stufe erster Ordnung und $S$ die Menge der geschlossenen Formeln von L. Ein Herbrand-Modell für S ist eine Herbrand-Interpretation für L, die ein Modell für S ist [51].

**Backtracking.** In Prolog wird als Auswertungsstrategie die Tiefensuche verwendet [33]. Bei der Beweissuche kann es vorkommen, dass das Ziel mit mehr als einer Regel übereinstimmt. Dies wird als Auswahlpunkt bezeichnet. Wenn ein Auswahlpunkt erreicht wird, wählt man die erste unter den übereinstimmenden Regeln aus, in der Reihenfolge, in der die Regeln präsentiert wurden. Wenn dieser Versuch eines Beweises fehlschlägt, kehrt man zu den vorhergehenden Auswahlpunkten zurück. Dieser Vorgang wird Backtracking (dt. Rücksetzverfahren) genannt [60].

Die syntaktischen und semantischen Eigenschaften von Constraints werden im nächsten Kapitel erläutert.

## 2.2 Constraints

### 2.2.1 Repräsentation und Semantik von Constraints

In einem typischen Constraint-Programm wird versucht, eine Lösung oder Lösungen zu dem in dem Programm modellierten Problem zu finden. Ein Problem, das in einem Constraint-Programm modelliert wurde, besitzt Variablen mit Domänen, aus denen die Variablen Werte annehmen, und Constraints, die die Domänen der Variablen einschränken [28]. Eine Lösung des Problems entspricht dann einer gültigen (oder optimierten/besten) Variablenbelegung, sodass die Constraints erfüllt sind. Constraints in Constraint-Programmie-

rung sind besondere Prädikate, die erfüllt werden müssen [30]. Genauer gesagt, jedes Constraint spezifiziert eine Relation in einer Untermenge von Objekten eines Problems, wo jedes Objekt eine Menge von möglichen Werten hat. So beschränkt ein Constraint den Raum der möglichen Werte für solche Objekte [63].

Im Vergleich zu der üblichen Logikprogrammierung lässt das CLP(X)-Schema Berechnungen direkt auf der gewünschten Domäne zu [78]. Zum Beispiel würde man ein Optimierungsproblem mit ganzzahligen Variablen gerne auf einem ganzzahligen Intervall modellieren und dieses Intervall direkt als die Domäne für die Berechnung nutzen, was in der Logikprogrammierung nicht möglich wäre. Dafür werden besondere Funktionen und Prädikatensymbole, die feste Interpretationen in der Domäne der Berechnung besitzen, eingeführt. Angenommen, man wählt die Menge der ganzen Zahlen als die Variablendomäne. Dann müssen auch Symbole wie $<, \leq, \geq, >, =, \neq$ eingeführt werden, die jeweils eine bestimmte Bedeutung in der Menge der ganzen Zahlen haben. Die Formel mit diesen besonderen Funktionen und Prädikatsymbolen nennt man „Constraintformel“. Informell wird oft das Wort „Constraint“ statt „Constraintformel“ verwendet [28].

Constraints haben die Form $c(x_1,..,x_n)$ mit einer endlichen Anzahl von Variablen $x_1,..,x_n$. Jede Variable $x_j$ kann einen Wert $v_j$ von einer endlichen Menge $D_j$, die die Domäne von $v_j$ heißt, haben [10]. Constraints können unterschiedliche Stelligkeit haben. Sei zum Beispiel A eine Variable über der Domäne der natürlichen Zahlen, so ist ein Constraint der Form $A \leq 3$ ein unäres Constraint, weil das Constraint nur A als Variable hat. Analog könnte man auch binäre Constraints definieren. Seien $B$ und $C$ Variablen, dann würde ein Ausdruck wie $3 * B \geq C$ einem binären Constraint entsprechen [43].

Es kann auch solche Fälle geben, in denen man alle Variablen einer Menge im gleichen Sinne beschränken möchte. Ein Beispiel dazu könnte folgendermaßen aussehen: Angenommen, seien vier Variablen $A, B, C, D$ definiert auf derselben Domäne, sodass alle Variablen ganzzahlige Werte im Intervall $[0, 10]$ nehmen können. Sei als ein Constraint gewünscht, dass alle Variablen unterschiedliche Werte haben. Dies lässt sich zwar mit binären Constraints formulieren, doch dafür wären insgesamt sechs Constraints notwendig, nämlich $A \neq B, A \neq C, A \neq D, B \neq C, B \neq D, C \neq D$. Diese Darstellung bedeutet eine komplette Suche für jede Variable auf ihrer Domäne, und ist daher entsprechend ineffizient.

Diese aufwändige Suche kann mit globalen Constraints vereinfacht werden. Ein globales Constraint ist ein Constraint, welches eine Beziehung zwischen einer nicht vorab definierten Anzahl von Variablen darstellt. Ein Beispiel für globale Constraints ist das *alldifferent*-Constraint ($alldifferent(x_1,...,x_n)$ mit $n \in \mathbb{N}$), das besagt, dass die Werte der Variablen $(x_1,...,x_n)$ paarweise verschieden sein müssen. Es ist einfach zu sehen, wie das *alldifferent*-Constraint in dem obigen Beispiel nützlich sein kann. Mit globalen Constraints kann man nicht nur die Anzahl der nötigen Constraints reduzieren, sondern das Suchproblem lässt sich auch effizienter lösen, zum Beispiel mit Hilfe von Graphenalgorithmen [63].

Ein nützliches Werkzeug, um mit globalen Constraints arbeiten zu können, ist „Global Constraint Catalog“ von Nicolas Beldiceanu, Mats Carlsson und Jean-Xavier Rampon. „Global Constraint Catalog“ ist eine Sammlung von globalen Constraints, die sowohl in Form einer Web-Seite [7] als auch in Form eines technischen Berichts verfügbar ist [8]. Dieser Katalog enthält nicht nur die Dokumentation von globalen Constraints, sondern auch Anwendungsbeispiele und Algorithmen zum Lösen der Constraints. Im Jahr 2022 sind insgesamt 423 globale Constraints im „Global Constraint Catalog“ gelistet.

Ein Constraint-Programm ist syntaktisch eine Sammlung von Klauseln, die entweder Regeln oder Fakten sind. Regeln in Constraint-Programmierung können in ihren Prämissen auch Constraints enthalten. Damit ist die Prämisse einer Regel eine Konjunktion von Constraints und Atomen.

Das folgende Beispiel [16] ist für das Verständnis der Syntax von Constraint-Programmen hilfreich. Sei die Definition einer Mahlzeit gegeben, die aus einer Vorspeise, einem Hauptgericht und einer Nachspeise besteht. Damit diese Mahlzeit leicht ist, wird vorausgesetzt,

dass die Summe von Brennwerten den Wert 10 nicht überschreitet.

```
lightmeal(A,M,D) :-
     I > 0, J > 0, K > 0,
     I + J + K <= 10,
     appetiser(A,I),
     main(M,J),
     dessert(D,K).
```

```
meat(beef,5).
```

Der linke Ausdruck ist ein Beispiel für eine Regel. Dabei ist `lightmeal(A,M,D)` der Kopf und der Teil nach dem Zeichen „:-“ ist der Rumpf der Regel. Der Rumpf ist eine Konjunktion von Constraints `I > 0, J > 0, K > 0, I + J + K <= 10` und Atomen `appetiser(A,I),main(M,J),dessert(D,K)`. Logisch betrachtet bedeutet dieser Ausdruck, dass `lightmeal(A,M,D)` erfüllt ist, falls der Regelrumpf und damit auch alle Constraints und Atome erfüllt sind. Der rechte Ausdruck ist ein Beispiel für einen Fakt. Fakten besitzen keinen Rumpf und haben diese einfache Form.

Die linke Regel hat dabei folgende Bedeutung: Eine Mahlzeit bestehend aus den Nahrungsmitteln *A,M,D* ist eine leichte Mahlzeit, wenn *A* eine Vorspeise ist (mit einem positiven Brennwert *I*), *M* eine Hauptgericht ist (mit einem positiven Brennwert *J*), *D* eine Nachspeise ist (mit einem positiven Brennwert *K*) und $I + J + K$ kleiner oder gleich 10 ist. Der Fakt rechts im Beispiel besagt, dass der Brennwert von Rindfleisch 5 ist.

Konzeptuell ist die operationale Semantik der Constraint-Programmierung eine Verallgemeinerung der Semantik der Logikprogrammierung. In einem Resolutionsschritt der Logikprogrammierung muss das Vorhandensein eines allgemeinsten Unifikators zwischen den ausgewählten Teilzielen und dem Kopf der Klausel überprüft werden. In der Constraint-Programmierung muss zusätzlich auch die Konsistenz der aktuellen Menge der Constraints (Constraintstore) mit den Constraints im Rumpf der Klausel überprüft werden. Es sind also zwei Solver beteiligt: Unifikation, wie in der Logikprogrammierung, und der spezifische Constraintlöser (siehe Kapitel 2.2) [32].

Die Semantik der Constraint-Programmierung kann als ein zielgerichtetes Ableitungsverfahren aus dem Ausgangsziel unter Verwendung der Programmklauseln beschrieben werden [39]. Ein Berechnungszustand besteht aus zwei Teilen:

1. Der Zielteil entspricht der Konjunktion der zu lösenden Ziele.
2. Das Constraintstore bezeichnet die Menge der bislang gesammelten bzw. hergeleiteten Constraints [39].

Sei $\langle G, S \rangle$ ein solcher Berechnungszustand, wobei $G$ der aktuelle Zielteil und $S$ das aktuelle Constraintstore ist. Constraint-Programme werden ausgeführt, indem die Ziele im Berechnungszustand anhand der Fakten und Regeln reduziert werden. In jedem Berechnungsschritt muss zusätzlich das Constraintstore konsistent bleiben. Konsistenz bezeichnet dabei die Existenz mindestens einer Variablenbelegung, sodass alle Constraints erfüllt sind [55].

Die gesamte Berechnung ist erfolgreich (d.h. eine Lösung wurde gefunden), falls von einem Anfangszustand $\langle G, true \rangle$ der Zustand $\langle G\prime, S \rangle$ hergeleitet werden kann, wobei $G\prime$ den leeren Zielteil darstellt und $S$ erfüllbar ist. Am Anfang wird das Constraintstore mit dem Wert „true“ initialisiert, weil die Variablenbelegungen von keiner Einschränkung betroffen sind [32].

Die Aufgabe eines Constraintlösers ist die Bestimmung einer Variablenbelegung, sodass alle Constraints erfüllt werden. Für viele Alltagsanwendungen ist es jedoch möglich, dass es keine Variablenbelegung gibt, die alle Constraints erfüllt. Das ist meistens der Fall, wenn man mit Constraints gewünschte Eigenschaften anstatt strikter Voraussetzungen formalisieren will. Constraints stehen dann für Präferenzen, deren Nichterfüllung soweit wie möglich vermieden werden soll. Diese Constraints werden weiche Constraints (engl. soft

constraints) genannt und können mehrere Ebenen der Erfüllbarkeit haben, die über die beiden Extreme Erfüllbarkeit bzw. Nicht-Erfüllbarkeit hinausgehen [32]. Diese Abstufungen können zum Beispiel mit Hilfe von numerischen Werten implementiert werden, wobei eine höhere Präferenz einen größeren Erfüllbarkeitswert und eine niedrigere Präferenz einen kleineren Erfüllbarkeitswert bedeutet [55].

### 2.2.2 Typische Domänen der Constraint-Programmierung

Das Ziel des von Jaffar und Lassez [42] entwickelten CLP(X)-Schemas war es, eine aussagekräftige und flexible Sprache zu schaffen. Flexibilität bedeutet hier die Freiheit bei der Wahl der Datentypen jener Variablen, die durch das Programm manipuliert werden. Je nach Anwendung kann es sein, dass mit booleschen Ausdrücken, ganzen Zahlen, Gleitkommazahlen oder Mengen gearbeitet werden soll. Dies war jedoch vor der Einführung des CLP(X)-Schemas in der Logikprogrammierung kaum möglich, denn alle Daten werden als uninterpretierte Strukturen (als endliche Bäume) kodiert [28].

Constraint-Programmierung ist keine Programmiersprache, sondern ein Programmierparadigma, also es gibt eine Vielzahl von Sprachen, die sich durch ihre Funktionalität, insbesondere ihre Domänen, unterscheiden [32]. Jede Domäne besitzt eigene algebraische Rechenoperationen, wie etwa der Schnitt bei Mengen oder die Addition bei arithmetischen Ausdrücken. Analog existieren auch Relationen, die ebenfalls spezifisch für die Domäne sind, wie die „Element"-Relation bei den Mengen oder Gleich-/Ungleichheit bei den arithmetischen bzw. booleschen Ausdrücken.

**Constraintlöser.** Die Einführung von Constraints in die Logikprogrammierung erfordert einen zusätzlichen Mechanismus, damit Constraits gelöst werden können. Im Kapitel „Verwandte Arbeiten" wurde für Prolog II Ungleichheit als ein primitives Constraint vorgestellt und im gleichen Sinne kann auch die Gleichheit als ein einfaches Constraint verwendet werden. Im Fall der Gleichheit kann für die Lösung Unifikation verwendet werden [28]. Dies gelingt jedoch nicht immer. Deswegen muss Unifikation mit einem Entscheidungsverfahren ergänzt werden, das besagt, ob ein Constraint oder eine Menge von Constraints erfüllbar ist. Dieser Mechanismus wird Constraintlöser genannt [28].

Domänen für CP-Anwendungen müssen einige Kriterien erfüllen: Zum einen muss die Domäne anwendungsrelevant sein, zum anderen muss für die Domäne ein vollständiger und effizienter Constraintlöser existieren. Ein Constraintlöser ist vollständig, falls er die Erfüllbarkeit von jedem beliebigen Constraint auf der Berechnungsdomäne entscheiden kann [28]. Erfüllbarkeit einer Menge von Constraints kann in manchen Domänen ein unentscheidbares Problem sein oder eine exponentielle Komplexität haben [68]. Dies beschränkt vollständige Löser zu relativ gut strukturierten Domänen wie etwa lineare arithmetische Terme oder zu Spezial-Löser wie etwa vollständige Boolean-Löser [68]. Aus Gründen der Effizienz soll ein Constraintlöser inkrementell sein, das heißt, wenn ein neues Constraint zu einer Menge von schon gelösten Constraints hinzugefügt wird, soll der Löser bereits existierende Teillösungen weiterhin berücksichtigen (siehe Kapitel 3.1) [28].

Im Folgenden werden drei der wichtigsten Constraintdomänen, arithmetische, endliche (engl. finite) und boolesche Domänen, ausführlicher untersucht.

#### Lineare arithmetische Domäne

Eine der Hauptmotivationen für die Kombination von Logikprogrammierung und Constraints war die mögliche Einführung von arithmetischen Constraints in den Kontext der Logikprogrammierung, denn obwohl Prolog Mechanismen zur Evaluierung arithmetischer Ausdrücke besitzt, entsprechen sie nicht den Erwartungen [30]. Eine Gleichung in der Form $X - 3 = Y + 5$ kann mit Prolog nicht behandelt werden. Aus der Sicht von Prolog

ist der Term $X - 3$ nicht gleich dem Term $Y + 5$, weil die einzigen Strukturen in Prolog uninterpretierte Strukturen sind.

In diesem Sinne wurde CLP(R) als eine Instantiierung des CLP(X)-Schemas mit der Domäne der reellen Zahlen entwickelt. CLP(R) ist die erste CP-Sprache, in der arithmetische Constraints über den reellen Zahlen eingeführt wurden. Der Constraintlöser von CLP(R) ist vollständig für lineare arithmetische Constraints. Nichtlineare Constraints werden zuerst als erfüllbar angenommen und verzögert (für eine gewisse Zeit nicht berücksichtigt), bis sie linear werden [28]. Verzögerte, nichtlineare Constraints werden erst dann ausgewertet, wenn eine ausreichende Anzahl ihrer Variablen bestimmt wurde, um sie linear zu machen. Erst nachdem diese Constraints linear sind, wird die Erfüllbarkeit dieser Constraints geprüft [36].

Lineare Constraints haben Anwendungen in Entscheidungsunterstützungssystemen, geometrischen Problemen [28] und bei der Analyse und Synthese analoger Schaltungen [36]. Constraints in dieser Domäne entsprechen kontinuierlichen Problemen, bei denen es eine unendliche Anzahl von Punkten im Suchraum zu untersuchen gibt [30]. Zudem lässt sich das nächste Element zu einem beliebigen Element in der Domäne nicht bestimmen.

Lineare arithmetische Ausdrücke sind Terme bestehend aus Zahlen, Variablen und aus den üblichen arithmetischen Operatoren Vorzeichenwechsel $(-)$, Addition $(+)$, Subtraktion $(-)$, Multiplikation $(*)$ und Division $(/)$. Damit die Linearität nicht verletzt wird, darf bei der Multiplikation höchstens ein Multiplikator eine Variable sein, und bei der Division muss der Teiler eine Zahl ungleich null sein. Ein lineares arithmetisches Constraint ist ein Ausdruck der Form $k_1 R k_2$, wobei $R$ eines der folgenden 2-stelligen Prädikate ist: $>, \leq, =, \geq, >, \neq$ [28].

Es existieren mehrere Entscheidungsverfahren, um ein System von linearen arithmetischen Constraints zu lösen. Ein Beispiel dafür ist eine Kombination von Gaußschem Eliminationsverfahren und einem modifizierten Simplex-Verfahren. Das Simplex-Verfahren ist ein Optimierungsverfahren aus der Numerik zur Lösung linearer Optimierungsprobleme [28].

## Endliche Domänen

Eine wichtige Klasse von Constraints ist auf endlichen Domänen definiert. Constraint-Programmierung nach diesem Schema heißt CLP(FD). Constraints dieser Klasse sind meistens arithmetische Constraints auf Variablen aus endlichen Mengen ganzer Zahlen (meistens Intervalle) [32]. Im Fall eines endlichen Intervalls der ganzen Zahlen werden Elemente der Domäne meistens in aufsteigender Reihenfolge iteriert. Eine andere Variablendomäne könnte eine endliche Menge von Buchstaben sein, die in lexikographischer Reihenfolge iteriert werden kann [59].

Eine Eigenschaft der CLP(FD)-Sprachen ist, dass sie Syntax zur Definition der Wertebereiche der Variablen bereitstellen müssen. Sei zum Beispiel die Variable A und das Intervall mit den ganzen Zahlen von 10 bis 20 gegeben, wobei A Werte aus diesem Intervall annimmt. Um das in einem Programm auszudrücken, existieren je nach der gewählten CLP(FD)-Sprache verschiedene syntaktische Schreibweisen, etwa: `A in [10..20]`, `A::[10,20]` oder `fd_domain(A,10,20)` [32].

Eine große Anzahl von CLP(FD)-Sprachen wurden mit akademischen oder kommerziellen Zielen entwickelt. Die erste Implementierung dieser Constraintdomäne in einer CP-Sprache war die von Pascal Van Hentenryck et al. in den späten 1980er-Jahren entwickelte Sprache CHIP. Seither wurden mehrere andere CLP(FD)-Sprachen entwickelt, wie etwa Eclipse, CIAO oder B-Prolog [32].

Anwendungsgebiete der endlichen Domäne sind zahlreich und umfassen: Planung, Testmustererstellung, Finanzentscheidungen, Frequenzzuteilung, Schaltungs-/Hardwaresimulationen und viele mehr [59].

Answer-Set-Programmierung (ASP) beschäftigt sich mit ähnlichen Problemen wie die Constraint-Programmierung mit der Constraintdomäne CLP(FD) [32]. Es gibt Arbeiten, die beide miteinander vergleichen [21, 53] oder auch beide integrieren [6].

### Boolesche Domäne

Terme dieser Domäne sind boolesche Terme. Diese bestehen aus Wahrheitswerten (richtig und falsch oder auch 0 und 1) sowie Variablen aus der booleschen Domäne und logischen Verknüpfungen (z.B. $\vee, \oplus\ (XOR), \wedge, \neg,$) [28]. Das einzige Constraint zwischen booleschen Termen ist Gleichheit bzw. Ungleichheit [28]. In manchen Implementierungen (z.B. CHIP) ist es möglich, zusätzliche Konstanten in Termen zu verwenden. Dies ist besonders für Anwendungen in Hardware-Verifikation wichtig, weil diese Konstanten verwendet werden können, um symbolische Namen für Eingangsargumente von Schaltungen darzustellen [30].

Beim Lösen boolescher Constraints werden verschiedene Methoden umgesetzt. In Fall von CHIP wird ein boolescher Unifikationsalgorithmus verwendet [28]. Einige wichtige Anwendungsgebiete der booleschen Constraints sind Schaltungsdesign, Hardware-Verifikation und Theorembeweise [28]. Ein Beispiel für Hardware-Verifikation wird später ausführlicher behandelt (siehe Kapitel 5.3).

## 2.2.3 Constraint Propagierung

Constraintdomänen erweitern die Logikprogrammierung um die Möglichkeit, Lösungen abhängig von einer Domäne einzuschränken. Gleichzeitig eröffnet dies jedoch auch die Möglichkeit, nach Lösungen effizienter suchen zu können. Mit diesem Suchverhalten beschäftigen sich dieses und das nachfolgende Kapitel.

Bei der Suche nach einer Lösung wird in Logikprogrammierung eine einfache Tiefensuche eingesetzt, die nach der Generieren-und-Testen-Heuristik (engl. generate and test) funktioniert [28]. Diese Methode besitzt aber Leistungsprobleme, besonders bei großen Suchanwendungen. Deswegen wurden Techniken entwickelt, um die Suchprozesse intelligenter zu machen. Der Begriff Konsistenztechniken (engl. consistency techniques) beschreibt diese Bemühungen [28].

Konsistenztechniken wurden vor der Einführung der Constraint-Programmierung in der Forschung im Bereich der künstlichen Intelligenz untersucht. Mit dem Ziel, das Suchverhalten der Logikprogrammierung zu verbessern, schlug Hervé Gallaire 1985 den Einsatz von Konsistenztechniken in der Logikprogrammierung vor [31]. Die Idee war, Constraints zu benutzen, um den Suchraum zu Beginn der Suche einzuschränken, anstatt zuerst Ergebnisse zu berechnen, die anschließend durch Constraints eingeschränkt werden. Das Paradigma Generieren und Testen wurde mit diesem Ansatz durch Beschränken und Generieren ersetzt [32].

Eine Lösung zu einem Problem, das in einer Sprache der Constraint-Programmierung modelliert wurde, wird also in zwei Schritten gefunden, Constraint Propagierung (engl. constraint propagation) und Suche. In der Constraint Propagierung ist es das Ziel, den Suchraum zu verkleinern. Die Suche selbst wird auf diesem verkleinerten Suchraum durchgeführt. Constraint Propagierung und Suche wechseln einander ab, bis eine Lösung bzw. deren Abwesenheit festgestellt wird [47]. In diesem Kapitel werden vier Propagierungsmethoden vorgestellt: Bogen-Konsistenz, Pfad-Konsistenz, Grenzen-Konsistenz und Brechen der Symmetrie.

**Bogen-Konsistenz.** Die erste Propagierungsmethode, die betrachtet werden soll, ist die Bogen-Konsistenz (engl. arc consistency). Diese Konsistenztechnik eliminiert inkonsistente Werte unter Berücksichtigung eines Constraints und Variablen, deren mögliche Werte

durch dieses Constraint beschränkt sind [24]. Seien zum Beispiel die Variablen mit einer endlichen Domäne gegeben durch: $A,B,C,D,E,F :: [1,5]$. Außerdem sind die folgenden Constraints für diese Variablen gegeben: $A < B, A < C, B < F, C < E, D < E, E < F, B \neq C$. Aus dem Constraint $A < B$ können folgende Schlussfolgerungen gezogen werden: Die Domäne von $A$ enthält den Wert 5 aber wegen dem Constraint $A < B$ kann die Variable $A$ mit diesem Wert nicht belegt werden. Analog dazu kann $B$ den Wert 1 nicht annehmen. Daraus ergeben sich die folgenden reduzierten Domänen für $A$ und $B$: $A :: [1,4]$ und $B :: [2,5]$.

Bogen-Konsistenz angewandt auf die oben dargestellten Variablen und Constraints ergibt die folgenden Domänen für einzelne Variablen:

$$A :: [1,2], B :: [2,4], C :: [2,3], D :: [1,3], E :: [3,4], F :: [4,5]$$

Konsistenztechniken sind meistens nicht ausreichend, um ein Problem zu lösen, weil auch nach der Propagierung inkonsistente Kombinationen von Werten in den resultierenden Domänen zu finden sind [28]. Das ist auch sichtbar in den resultierenden Domänen aus dem letzten Beispiel. Obwohl die Bogen-Konsistenz auf die oben dargestellten Variablen und Constraints angewandt wurde, können immer noch solche Variablenbelegungen gewählt werden, die ein Constraint nicht erfüllen: Zum Beispiel sind $A = 2$ und $B = 2$ nach der Domänenbeschränkung wählbar, aber diese Kombination erfüllt das Constraint $A < B$ nicht.

Um eine Lösung zu finden, muss eine Suche durchgeführt werden, in der eine Variable mit einem Wert belegt wird. Damit wird es möglich, nochmal Propagierung durchzuführen. Angenommen, im Beispiel wird $A$ mit dem Wert 2 belegt. Dann ergeben sich wegen der Constraints $A < B, A < C$ die folgenden Domänen: $B :: [3,4], C :: [3]$. Erst ab diesem Punkt kann das Constraint $B \neq C$ angewendet werden. Damit wird die Propagierung fortgesetzt und es ergeben sich folgende Domänen für die jeweiligen Variablen:

$$A :: [2], B :: [4], C :: [3], D :: [1,2,3], E :: [4], F :: [5]$$

Wie ein Ungleichheitsconstraint mit Bogen-Konsistenz behandelt wird, wird im nächsten Beispiel gezeigt. Angenommen seien zwei Variablen $X$ und $Y$. Außerdem haben beide Variablen Domänen mit mehr als einem Element. Dann kann das Constraint $X \neq Y$ keine neue Information liefern, weil es für jedes Element in der Domäne von X ein konsistentes Element in der Domäne von Y geben kann. Damit Bogen-Konsistenz für dieses Constraint verwendet werden kann, muss eine Domäne die andere Domäne einschränken. Das heißt, Bogen-Konsistenz für das Ungleichheitsconstraint lässt sich ziemlich einfach und effizient implementieren, aber kann nur in besonderen Fällen nützlich sein.

Die Bogen-Konsistenz stößt auch schnell an ihre Grenzen. Seien die Variablen und ihre Domänen folgendermaßen gegeben: $X,Y,Z :: [1,2]$. Ein Constraintlöser kann mit Hilfe der Bogen-Konsistenz nicht feststellen, dass die Constraints $X \neq Y, Y \neq Z, Z \neq X$ nicht erfüllbar sind [28].

**Pfad-Konsistenz.** Die zweite Konsistenztechnik, die betrachtet werden soll, ist die Pfad-Konsistenz (engl. path consistency) [32]. Diese ist eine aufwändige Methode, die selten in Constraintlöser eingebaut wird. Das Ziel der Pfad-Konsistenz ist das Ableiten neuer Information über das Problem, sodass jeweils ein Paar Constraints zusammen untersucht wird. Die resultierende Konsistenz ist stärker als die Bogen-Konsistenz [32]. Ein Beispiel dazu kann wie folgt aussehen: Seien die Variablen mit ihren Domänen $A,B :: [1,10]$ und die Constraints $A \leq B, B \leq A, A \neq B$ gegeben. Wie im vorigen Beispiel mit dem Ungleichheitsconstraint liefert die Bogen-Konsistenz hier keine Verkleinerung der Domänen. Wenn jedoch zwei Constraints gleichzeitig betrachtet werden, zum Beispiel $A \leq B \wedge B \leq A$, kann daraus die Gleichheit von A und B, also $A = B$, abgeleitet werden. Zusammen mit der Information $A \neq B$ folgt, dass das Problem keine Lösung hat.

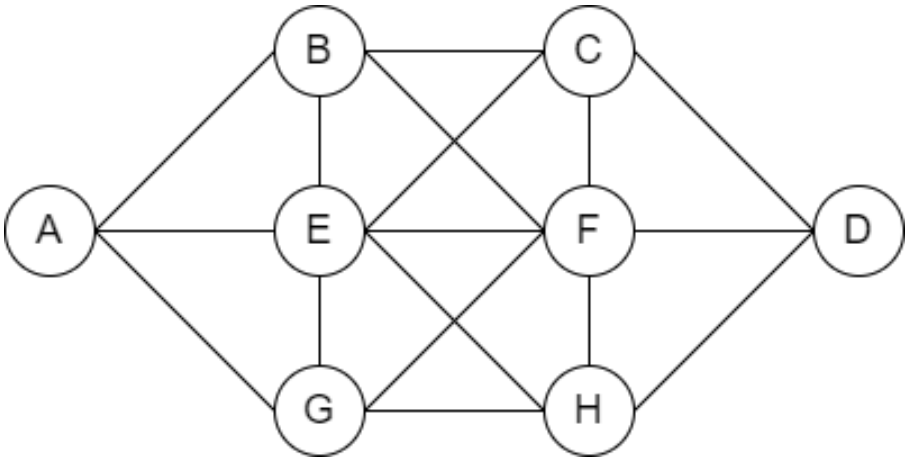


Abbildung 2.1: Layout von dem Crystal-Maze Problem

**Grenzen-Konsistenz.** Eine schwächere Alternative zur Bogen-Konsistenz ist die Grenzen-Konsistenz (engl. bound consistency) [32]. Die Grenzen-Konsistenz betrachtet nur die Maxima und Minima der Domänen, sodass eine kleinere Anzahl von Konsistenzprüfungen im Vergleich zur Bogen-Konsistenz nötig ist, aber sie liefert auch eine schwächere Propagierung. Aufgrund der mathematischen Regeln für Intervalle ist diese Methode besonders nützlich für die Prädikate $=, >, \geq, <, \leq$.

Als Beispiel seien die Variablen mit ihrer Domäne $X, Y, Z :: [1, 10]$ angenommen. Zudem sei das Constraint $3 * X + 2 * Y + 2 < Z$ für diese Variablen gegeben. Die Grenzen-Konsistenz verwendet Maxima und Minima der Intervalle, um inkonsistente Werte aus den Domänen zu entfernen. Das Maximum für $Z$ ist 10 und daraus folgt $3 * X + 2 * Y < 8$. Das Minimum von $X$ ist 1 und daraus folgt $2 * Y < 5$ und somit $Y :: [1, 2]$. Analog folgt aus $min(Y) = 1$, dass $3 * X < 6$ und folglich $X :: [1]$. Von den Minimumwerten (jeweils 1) kann für $Z$ gefolgert werden, dass $Z :: [8, 9, 10]$.

**Symmetrie.** Es kann sein, dass die zu lösenden Probleme symmetrische Teilaufgaben besitzen, sodass der Suchraum eingeschränkt werden kann. Deswegen wurden Techniken entwickelt, um die symmetrischen Teile des Suchraums zu entfernen. Der Überbegriff für solche Techniken ist Brechen der Symmetrie (engl. symmetry breaking) [32]. Dies umfasst Methoden wie Umschreibung des Constraint-Programms oder Hinzufügen neuer symmetrie-brechenden Constraints (entweder per Hand oder automatisch).

Wie die Symmetrie-Eigenschaft genutzt werden kann, um den Suchraum des Problems zu verkleinern, lässt sich mit dem folgenden Beispiel leicht zeigen. Die Abbildung 2.1 wurde in der britischen Spielshow „The Crystal Maze“ als ein Puzzle vorgestellt. Das Problem wurde danach von Patrick Prosser und Ian Miguel in [57] als ein Constrainterfüllbarkeitsproblem (engl. constraint satisfaction problem, siehe nächstes Kapitel) formuliert und stellt gleichzeitig ein gutes Beispiel dafür dar, wie die Symmetrie-Eigenschaft eines Problems ausgenutzt werden kann.

Das Ziel in diesem Problem ist, dass jeder Knoten in einem symmetrischen Graphen (vgl. Abbildung 2.1) mit einem Wert aus der Domäne $[1, 8]$ belegt wird. Zudem existieren die Constraints, dass jeder Wert nur einmal verwendet werden darf (*alldifferent*) und dass keine zwei Nachbarknoten zwei sukzessive Werte haben (z.B. falls der Knoten A mit dem Wert 2 belegt wird, ergibt sich für die Domäne des Knotens B: $B :: [4..8]$).

Die geometrische Symmetrie des in Abbildung 2.1 dargestellten Problems ist einfach zu sehen und diese geometrische Symmetrie impliziert auch eine Symmetrie im Constrainter-

füllbarkeitsproblem. Man beginnt die Lösung mit den Knoten E und F, weil diese Knoten die größte Anzahl von Nachbarknoten haben. Knoten E und F werden jeweils 6-mal von dem Constraint der Sukzessivität beschränkt. Mögliche Werte für diese Knoten sind somit $\{1,8\}$. Angenommen, der Knoten E sei mit 1 und der Knoten F mit 8 belegt. Dann ergeben sich für die Knoten die folgenden Domänen: $B,C,G,H \in \{3,4,5,6\}$, $A \in \{3,4,5,6,7\}$, $D \in \{2,3,4,5,6\}$. Nun könnte man den Knoten A mit 7 und den Knoten D mit 2 belegen, weil diese Werte in keinem anderen Knoten möglich sind. Dann wird die Propagierung weiter durchgeführt, um die Domänen der Knoten $B,C,G,H$ weiter zu beschränken, sodass sich die folgende Belegung am Ende als Lösung ergibt: $A = 7$, $B = 3$, $C = 5$, $D = 2$, $E = 1$, $F = 8$, $G = 4$, $H = 6$.

Jedoch ist dies nicht die einzige Lösung, weil mit der vertauschten Belegung der Knoten E und F eine weitere gültige Lösung entsteht. Das ist eine symmetrische Lösung, weil diese erste Entscheidung weitere Belegungen entsprechend ändert. Es ist möglich, diese umgekehrte Belegung gar nicht zu betrachten, da bekannt ist, dass aufgrund der Symmetrie des Problems eine symmetrische Lösung existiert. Falls aber auch diese alternative Lösung wünschenswert ist, kann diese mit nur wenig Arbeitsaufwand abgeleitet werden.

Somit lässt sich sagen, dass Symmetrie ein nützliches Werkzeug bei der Constraint Propagierung sein kann. Wie das Beispiel in Abbildung 2.1 auch zeigt, ist es jedoch notwendig, ein gutes Verständnis des Problems zu haben, um diese Eigenschaft ausnutzen zu können.

### 2.2.4 Auflösen von Constraints

Im vorherigen Kapitel wurde gezeigt, dass Constraint Propagierung alleine nicht ausreicht, um eine Lösung zu generieren. Oftmals ist das Ergebnis der Propagierung eine Reduktion der Variablendomänen. Wenn keine Propagierung mehr möglich ist, muss eine Suche durchgeführt werden [28]. Dafür werden Annahmen getroffen, in denen Variablen mit Werten belegt werden. Das wird als Beschriftungsverfahren (engl. labeling procedure) bezeichnet [24]. Mit den zusätzlich getroffenen Annahmen wird geprüft, ob zusätzliche Propagierungsschritte möglich werden. Diese zwei Schritte des Constraintlösens wiederholen sich dann gegenseitig, bis eine Lösung gefunden ist.

Nach getroffenen Annahmen existiert jeweils eine Verzweigung bezüglich der Nachfolgeschritte. Als Beispiel seien die Variablen mit ihrer Domäne $X,Y,Z :: [1,10]$ angenommen. Zusätzlich sei angenommen, dass sich nach einem Propagierungsschritt folgende Domänen für die Variablen ergeben: $X :: [1,5], Y :: [7,9], Z :: [2,5]$. Der Einfachheit halber werden Annahmen in lexikographischer Reihenfolge der Variablen gemacht und erst nach einer vollständigen Variablenbelegung wird eine Suche durchgeführt. Zum Beispiel sei die erste Annahme $X = 1$. Dann muss eine Annahme für $Y$ getroffen werden. Die Domäne von $Y$ hat 3 Elemente und jede dieser Elemente kann für die Annahme von Y gewählt werden. Das heißt, es existiert eine 3-stellige Verzweigung von $X = 1$ für alle mögliche Werte von $Y$. Analog sind für eine beliebige $Y$-Annahme vier $Z$-Annahmen möglich, sodass eine Baumstruktur entsteht und die Suche einer Baumsuche entspricht.

Die Baumsuche, die verwendet wird, ist oftmals eine Tiefensuche. Während dieser Suche wird auch das von der Logikprogrammierung bekannte Rücksetzverfahren (engl. backtracking) eingesetzt [24]. Dieses Verfahren erlaubt im Fall einer inkonsistenten Annahme, dass die problematische Variablenbelegung zurückgesetzt wird und die Suche ab dem letzten unproblematischen Stand fortgesetzt wird. Verwendung des Rücksetzverfahrens ist einer der Gründe, warum sich Constraint Propagierung gut in der Sprache Prolog implementieren lässt [28].

Es existieren auch zahlreiche Methoden und Heuristiken, um die Effizienz des Rücksetzverfahrens zu erhöhen, damit Lösungen schneller gefunden werden können. Diese Methoden können grob anhand folgender Aspekte beschrieben werden:

1. Welche Variable soll als nächstes belegt werden?
2. In welcher Reihenfolge sollen Werte von Variablen geprüft werden?
3. Können unvermeidliche Fehlschläge frühzeitig erkannt werden?

In vielen Fällen ist es am effektivsten, die Variable mit der kleinsten Domäne als nächstes zu wählen. Dieses Prinzip wird als „First-Fail-Prinzip“ bezeichnet [24]. Es wird erwartet, bei weniger Auswahlmöglichkeiten früher herauszufinden, ob diese richtig oder falsch sind.

Alternativ kann auch jene Variable gewählt werden, die in den meisten Constraints vorkommt. Somit kann mit einer Variablenbelegung eine möglichst große Anzahl von Variablendomänen verkleinert werden [28].

Eine Methode, um Fehler frühzeitig zu erkennen, ist die Vorwärtsprüfung (engl. forward checking). Nach dieser Strategie wird eine Bogen-Konsistenz zwischen Paaren von einer noch nicht mit einem Wert belegten und einer mit einem Wert belegten Variablen durchgeführt [5]. Als Beispiel seien die Variablen $T, X, Y, Z$ angenommen. Sei die Variable $T$ gewählt, sodass eine Annahme für diese Variable getroffen werden soll. Das heißt, $T$ muss mit einem Wert aus seiner Domäne belegt werden. Für die getroffene Annahme von $T$ wird Bogen-Konsistenz zwischen dieser Annahme und allen möglichen Werten der Variablen $X, Y, Z$ durchgeführt. Falls die Domäne einer Variable während dieser Konsistenzprüfung leer wird, kann mit der Annahme von $T$ keine Lösung generiert werden. Das heißt, die Suche wird fortgesetzt, indem die fehlgeschlagene Annahme mit dem Rücksetzverfahren zurückgesetzt wird. Damit stellt Vorwärtsprüfung sicher, dass für jede unbelegte Variable mindestens ein Wert existiert, der mit der aktuellen Variablenbelegung konsistent ist.

Constraintprobleme, in denen alle Lösungen zueinander gleichwertig sind, werden als Constrainterfüllbarkeitsprobleme (engl. constraint satisfaction problems) bezeichnet. Dies ist der Fall für alle bislang behandelten Beispiele. Eine Lösung zu einem Constrainterfüllbarkeitsproblem ist somit eine Wertezuweisung von Variablen, die alle Constraints erfüllt [5].

Lösungen zu einem Problem müssen jedoch nicht alle gleichwertig sein. Es ist möglich, eine beste oder optimale Lösung zu haben, was in der Praxis häufig vorkommt. Solche Probleme werden Constraintoptimierungsprobleme (engl. constraint optimisation problems) genannt. Probleme dieser Art besitzen zusätzlich eine Kostenfunktion, die minimiert werden soll. Neben dem Constraintlösen wird auch eine Optimierung durchgeführt. Jedes Constrainterfüllbarkeitsproblem kann als Constraintoptimierungsproblem aufgefasst werden, wenn zum Beispiel die Anzahl der Berechnungsschritte gezählt und minimiert wird [5].

KAPITEL 3

# Fortgeschrittene Techniken in Constraint-Programmierung

## 3.1 Inkrementelle Constraint-Programmierung

Systeme der Constrainterfüllbarkeit können als inkrementell oder nicht-inkrementell klassifiziert werden [58]. In nicht-inkrementellen Systemen führt das Finden einer neuen Lösung nach einer kleinen Änderung der Variablen oder Constraints zu einer vollständigen Neuberechnung durch den Löser. Das heißt, von der bisherigen Lösung wird kein Gebrauch gemacht. Inkrementelle Systeme hingegen behalten eine Reihe von Teilergebnissen bei. Jede Änderung wird mit der vorherigen Lösung kombiniert, um eine neue Lösung zu finden. Somit ist die inkrementelle Constraint-Programmierung eine wichtige Erweiterung der Constraint-Programmierung.

Ein Inkrement zu einem Constraintsystem kann als das Hinzufügen neuer Constraints oder Variablen verstanden werden [22]. Jedoch kann ein Inkrement je nach der Implementierung des Systems auch das Löschen von Constraints oder Variablen umfassen.

Dass sich die inkrementellen Techniken für Constrainterfüllbarkeit als nützlich erwiesen haben, ist in Systemen, die in [22, 74, 56] beschrieben sind, zu beobachten. Außerdem sind die Artikel „Incremental Constraint Satisfaction in Logic Programming“ [38] von Pascal Van Hentenryck und „Incremental Search in Constraint Logic Programming“ [74] von Pascal Van Hentenryck und Thierry Le Provost gute Quellen für das Thema, die neben der Implementierung auch den theoretischen Hintergrund der inkrementellen Constraint-Programmierung darstellen.

Inkrementelle Techniken sind besonders für jene Anwendungen geeignet, bei denen der Benutzer interaktiv Constraints angeben kann und sich von der Anwendung dennoch eine schnelle Reaktion erwartet [58]. Inkrementelle Löser sind schneller als nicht-inkrementelle, solange jede neue Lösung (d.h. Aktualisierung der Lösung, einschließlich Overhead) schneller ermittelt werden kann als eine vollständige Neulösung der gesamten Menge von Constraints. Dies erfordert beispielsweise, dass die Anzahl der Variablen, die nach einem Inkrement (Änderung) wieder aufgelöst werden müssen, deutlich kleiner ist als die gesamte Anzahl der Variablen im System.

Auf der anderen Seite können für gewisse Anwendungen nicht-inkrementelle Löser schneller sein. Um einmalige Lösungen für ein großes Constraintsystem zu finden, können nicht-inkrementelle Löser tatsächlich schneller sein, da diese keinen zusätzlichen Aufwand für die Verwaltung von Zwischenergebnissen der Lösungen hinnehmen müssen.

In der Praxis erweist sich das inkrementelle Lösen von Constraints als wünschenswert in mindestens drei Arten von Anwendungen [74]:

1. **Anpassung einer bestehenden Lösung zur Aufnahme neuer Informationen.** Diese Art der Anwendung ist typisch für Terminplanungsprobleme, die oft als Neuplanungsprobleme (engl. rescheduling) bezeichnet werden. Während der Ausführung des vorgeschlagenen Zeitplans können Probleme (z.B. Ausfall einer Maschine) auftreten, die eine Aktualisierung der bestehenden Lösung erfordern.

2. **Modifikation einer Lösung, um Prioritäten und Präferenzen hinzufügen.** Die Fahrplanplanung (engl. timetable scheduling) ist eine spezifische Anwendung, bei der der Benutzer möglicherweise die vom System vorgeschlagene Lösung nach seinen oder ihren eigenen subjektiven Kriterien einer guten Lösung ändern möchte.

3. **Interaktives Lösen von Problemen.** Anwendungen, bei denen eine Mensch-Maschine-Interaktion stattfindet und, die sich mit Constraints formulieren lassen.

Ein häufiges Anwendungsgebiet der inkrementellen Constraint-Programmierung ist der Bereich der Visualisierung. Visualisierung ist die Darstellung von Informationen in grafischer oder bildlicher Form [58]. Constraints sind nützliche Werkzeuge in der Visualisierung, denn die relativen Positionen von Objekten lassen sich mit Positionsconstraints (wie etwa „über“, „unter“, „berührend“) praktisch spezifizieren. Solche informellen Constraints werden in mathematische Formeln umgewandelt, um Constrainterfüllbarkeitsprobleme formulieren zu können. In einem derartigen Problem stellen Variablen die Positionen von Objekten dar, sodass eine Lösung des Problems zu einer absoluten Position des Objekts im angegebenen Bild führt.

Bewegung eines Objekts mit der Maus durch den Benutzer kann als ein Beispiel für Inkrementalität dienen. Es ist wünschenswert, dass sich das Objekt während der Aktion ständig bewegt, anstatt dass das Programm bis zum Ende der Bewegung wartet und erst dann die Koordinaten berechnet werden. Während der Bewegung ändert sich die Objektposition relativ zur vorherigen Position. Ein Ziel bei der Visualisierung ist, dass die Übergänge zwischen aufeinanderfolgenden inkrementellen Lösungen fließend sein sollen, so dass der Benutzer die Bewegung der Objekte leicht verfolgen kann [58].

Wie inkrementelle Constraints gelöst werden können, kann anhand eines Beispiels erklärt werden. Der beispielhafte Algorithmus heißt ICOLA (Incremental Constraint-based Object Layout Algorithm) [58]. ICOLA positioniert grafische Objekte anhand von Objektdeklarationen und Constraints, die die relativen Positionsbeziehungen zwischen den Objekten angeben. In der Erklärung des Beispiels werden Aspekte der Visualisierung möglichst abstrahiert, sodass der Schwerpunkt auf dem inkrementellen Constraintlösen liegt. Außerdem wird das Löschen in dieser Arbeit nicht betrachtet, obwohl ICOLA diese Aktion auch unterstützt.

Die Variablen des Constraintsystems seien mit $x, y, p, q$ dargestellt und stammen aus der Domäne der nicht-negativen natürlichen Zahlen. Diese Variablen stellen die absoluten Koordinatenpositionen der linken, rechten, oberen bzw. unteren Kante des Objekts dar. Außerdem werden $\alpha$ und $\gamma$ verwendet, um beliebige nicht-negative ganze Zahlen zu bezeichnen. Constraints werden in ICOLA als Formel der Form $x + \alpha \leq y$ dargestellt.

Sei $C$ die Menge der Constraints in einem derartigen Visualisierungssystem und $V$ die in $C$ vorkommenden Variablen. Eine Lösung $S$ ist dann eine Menge von Variablenbelegungen, so dass alle Constraints erfüllt sind.

Sei $G$ ein gerichteter und gewichteter Graph, sodass die Knoten des Graphs den Variablen in $V$ und die Kanten den Constraints $x + \alpha \leq y \in C$ entsprechen. Die Kante $x + \alpha \leq y$ verbindet Knoten $x$ und $y$ und hat das Gewicht $\alpha$. Eine Variable $y$ ist direkt von $x$ abhängig, falls $y \neq x$ und es existiert eine Kante von $x$ zu $y$ in G. Eine Variable $y$ ist von $x$ abhängig,

falls $y \neq x$ und es existiert ein Pfad von $x$ zu $y$ in G. Dieses Konzept der Abhängigkeit ist entscheidend für die Gestaltung von ICOLA. Insbesondere wird der Wert der Variablen $y$ erst dann in einer Berechnung verwendet, wenn der Wert jeder Variable $x$, von der $y$ abhängt (d.h. jede Variable $x$, so dass $x + \alpha \leq y \in C$ gilt) als korrekt bekannt ist.

**Konsistenzprüfung.** Vor dem Versuch, eine Lösung für ein Inkrement zu bestimmen, führt ICOLA zunächst eine Konsistenzprüfung durch, also eine Prüfung, ob das Inkrement zu einem konsistenten Constraintsystem führt. Die Constraints in $C$ sind konsistent, falls der Graph $G$ (durch die Constraints $C$ definiert) zyklusfrei ist.

Beim Hinzufügen eines Constraints $x + \alpha \leq y$ wird eine Konsistenzprüfung durchgeführt, indem im Graphen $G$ nach einem Pfad von $y$ nach $x$ gesucht wird. Wenn ein solcher Pfad existiert, würde das Inkrement zu einem Zyklus im Graphen führen, und das Inkrement ist unzulässig. Wenn kein solcher Pfad gefunden wird, führt das Hinzufügen von $x + \alpha \leq y$ zu $C$ nicht zu einer Inkonsistenz, und das Inkrement kann erlaubt werden. Es ist erforderlich, dass das Constraintsystem sowohl vor als auch nach jedem Inkrement konsistent ist.

Da das Constraintsystem in ICOLA stark unterbeschränkt (engl. under-constrained) ist, gibt es eine unendliche Anzahl von individuellen Lösungen. Um mit dem Problem der Unterbeschränkheit umgehen zu können, wurden in ICOLA zwei zusätzlichen Heuristiken umgesetzt:

1. Das Objekt wird möglichst in die linke obere Ecke platziert, sodass die Koordinatenwerte klein sind.

2. Objekte sind in ihrer Ausdehnung so klein wie möglich.

Die erste Heuristik genießt eine höhere Priorität als die zweite. Dies ist notwendig, damit der Constraintlöser eine eindeutige Lösung erzeugen kann.

**Inkrementelle Constrainterfüllbarkeit.** ICOLA behält nach der Problemlösung einen internen Zustand bei, der eine Menge von Variablenzuweisungen enthält, bei denen jede Variable den kleinstmöglichen Wert hat, der die Constraints noch erfüllt. Das heißt, der interne Zustand stellt eine Lösung für die Menge von Constraints dar, unter der ausschließlichen Berücksichtigung der ersten Heuristik. Der interne Zustand enthält daher wichtige Informationen über die Werte der Variablen, die in der endgültigen Lösung nicht verfügbar sind.

Bei der Ermittlung einer neuen Lösung kombiniert ICOLA das Inkrement mit der vorherigen Lösung und dem internen Zustand. Das Ergebnis ist eine Lösung für das neue Constraintsystem und ein neuer interner Zustand. Der interne Zustand wird in einer Datenstruktur ähnlich einer Kodierung des gerichteten, zyklusfreien Graphen G festgehalten. Eine Kodierung der vorherigen Lösung bleibt auch intern innerhalb des Solvers erhalten. ICOLA versucht, die Wiederverwendung sowohl der vorherigen Lösung als auch des internen Zustands zu maximieren. Das Ziel besteht darin, nur die Teile der Lösung und des internen Zustands zu ändern, die eine Aktualisierung erfordern.

ICOLA stellt somit dank seines Anwendungsgebietes Visualisierung ein verständliches Beispiel für die inkrementelle Constrainterfüllbarkeit dar. Viele bekannte Sprachen der Constraint-Programmierung unterstützen inkrementelle Constraint-Programmierung. CLP(R) enthält beispielsweise eine inkrementelle Version des Simplex-Algorithmus, während CHIP einen inkrementellen Löser für Constraints über endliche Domänen enthält [22].

## 3.2 Nebenläufige Constraint-Programmierung

Eine wichtige Erweiterung der Constraint-Programmierung ist die nebenläufige Constraint-Programmierung, bei der die Constraint-Programmierung mit der nebenläufigen Programmierung kombiniert wird [29]. Die Idee hinter diesem Ansatz besteht darin, die Auswertung von Constraints in Form von nebenläufigen Berechnungen auszudrücken [28]. Eine andere, neue Beobachtung war, dass Constraints nicht nur zur Lösung kombinatorischer Probleme, sondern auch zur Spezifikation der Prozesskommunikation und -synchronisation verwendet werden können [34]. Die Grundlagen dieses Konzepts wurden von Vijay A. Saraswat und Martin Rinard im Jahr 1989 [66] und von Vijay A. Saraswat im Jahr 1993 [65] eingeführt. Die Semantik der nebenläufigen Constraint-Programmierung hat ebenfalls Vijay A. Saraswat beschrieben [64].

Im Rahmen der nebenläufigen Constraint-Programmierung werden neben dem Constraintlösen auch andere Operationen an Constraints sowie zusätzliche Möglichkeiten, diese zu kombinieren, umgesetzt. Eine wichtige derartige Operation ist das Einfordern von Constraints (engl. constraint entailment) [39]. Dabei wird entschieden, ob ein Constraint aus der Konjunktion mehrerer Constraints impliziert wird. Angenommen, es existiere ein Constraintstore mit den Elementen $\{A+B \geq 1, A+(1-B) \geq 1\}$ für die Variablen $A$ und $B$ mit der Domäne $\{0,1\}$. Aus diesen beiden Constraints kann $A = 1$ abgeleitet werden. Algorithmen für dieses Verfahren sind Erweiterungen der Algorithmen zum Lösen von Constraints [28].

Ein wichtiges Werkzeug für die nebenläufige Constraint-Programmierung sind Wachen (engl. guards). Diese definieren eine logische Bedingung in Form eines booleschen Ausdrucks, der als wahr bewertet werden muss, damit die Programmausführung in dem betreffenden Zweig fortgesetzt werden kann. Wachen sind kein neues Konzept der nebenläufigen Constraint-Programmierung, sondern werden in anderen Programmiersprachen ebenso verwendet. Sei als Beispiel ein und-Constraint in der Syntax von Constraint Handling Rules gegeben. Um den Kopf und den Rumpf einer Regel zu trennen wird das <=>-Symbol verwendet. Seien X und Y die booleschen Eingaben und sei Z die Ausgabe. Dabei sind die Wachen und Ziele mit einem „|“ getrennt. Die Vereinfachungsregeln für dieses Constraint in Constraint Handling Rules würden dann wie folgt aussehen [28]:

```
and(X,Y,Z) <=> X = 0 | Z = 0
and(X,Y,Z) <=> Y = 0 | Z = 0
and(X,Y,Z) <=> Z = 1 | X = 1, Y = 1
and(X,Y,Z) <=> X = 1 | Y = Z
and(X,Y,Z) <=> Y = 1 | X = Z
and(X,Y,Z) <=> X = Y | Z = X
```

Wird der dritte Ausdruck betrachtet, das Constraint $and(X,Y,Z)$ kann als $X = 1, Y = 1$ vereinfacht (umgeschrieben) werden, nur falls es bekannt ist, dass der Wert von $Z$ 1 ist. Aus diesem Beispiel kann beispielsweise auch schlussgefolgert werden, dass die Bedingung $Z = 0$ nicht ausreichend wäre, um neue Konsequenzen zu extrahieren. Der Grund dafür ist, dass $Z = 0$ alleine keine Wache erfüllt.

Nebenläufige Programmierung bezieht sich auf das Szenario, dass mehrere Prozesse miteinander kommunizieren müssen. Dafür existiert es zwei Implementierungsmöglichkeiten [73]: Entweder die Prozesse verfügen über eine gemeinsame Ressource (z.B. Speicher) oder sie können sich gegenseitig Nachrichten senden.

Logikprogramme sind im Allgemeinen zur parallelen Ausführung gut geeignet. Es existieren folgende parallele Auswertestrategien für Logikprogramme [35]:

1. **UND-Parallelismus:** Mehrere Literale im Rumpf einer Anfrage werden parallel abgearbeitet.

2. **ODER-Parallelismus:** Es können mehrere Programmklauseln für ein Literal existieren. Die Klauseln für dasselbe Literal werden für einen Beweis parallel verwendet.

3. **Engagierte Wahl (engl. committed choice):** Klauselrümpfe werden um eine Wache erweitert.

   ```
   <head> :- <guard> | <body>
   ```

   Falls mehrere Klauseln für ein Literal existieren, wird nicht-deterministisch eine Klausel ausgewählt, bei der die Wache beweisbar ist, und die anderen Möglichkeiten ignoriert.

In den ersten beiden Möglichkeiten ist die Parallelität nicht unmittelbar sichtbar für den Programmierer, sondern sie ist nur ein Implementierungsaspekt. Die engagierte Wahl enthält jedoch explizite Konstrukte zur Nebenläufigkeit. Diese Konstrukte wurden erstmalig im Rahmen von Japans „Fifths Generation Computing Project“ entwickelt [39].

Die Ideen von diesem Projekt wurde von Vijay A. Saraswat weiterentwickelt und mit der damals neu eingeführten Constraint-Programmierung kombiniert. Das Ergebnis war das Modell für die nebenläufige Constraint-Programmierung [66]. Nach diesem Modell existieren mehrere nebenläufige Prozesse, die auf einem gemeinsamen Constraintstore arbeiten. Genauso wie CLP(X) kann das Modell für die nebenläufige Constraint-Programmierung mit unterschiedlichen Strukturen je nach dem Typ der Constraints in dem Constraintstore instanziiert werden.

Jeder nebenläufige Prozess hat zwei Aktionsmöglichkeiten: *ask* und *tell* [73]. Mit der *ask*-Operation kann gefragt werden, ob die Constraints im Constraintstore das angefragte Constraint implizieren (engl. entail). Mit der *tell*-Operation können neue Constraints zum Constraintsore hinzugefügt werden. Ein neues Constraint kann jedoch nur nach einer Konsistenzprüfung zum Constraintstore hinzugefügt werden, weil ein Constraintstore keine inkonsistenten Constraints enthalten darf. Unter Konsistenz wird wie bei der normalen Constraint-Programmierung die Existenz mindestens einer Variablenbelegung verstanden, die alle Constraints erfüllt. Das Hinzufügen eines Constraints wird als (logische) Konjunktion der hinzugefügten Constraint und des Inhalts vom Constraintstore, der den vorherigen Zustand repräsentiert, dargestellt. Bei der Aktualisierung des gemeinsamen Speichers geht es also nicht darum, die Werte der Variablen zu ändern, sondern darum, einige der zuvor möglichen Werte auszuschließen. Mit anderen Worten: Das Constraintstore wird monoton verfeinert [29].

Die Synchronisierung zwischen nebenläufigen Prozessen wird durch eine blockierende Ask-Operation erreicht [66]: Ein Prozess wird blockiert, wenn er versucht zu fragen, ob ein Constraint durch das gegebene Constraintstore impliziert ist. Falls dies noch nicht der Fall ist, bleibt der Prozess so lange blockiert, bis (wenn überhaupt) andere gleichzeitig ausführende Prozesse dem Constraintstore Constraints hinzufügen, sodass das resultierende Constraintstore die Anfrage oder ihre Negation implizieren (engl. entail) kann. Somit erfolgt die Synchronisation durch Gültigkeit von Constraints, also, auf einer logischen Ebene [35].

Wie mehrere Prozesse über ein gemeinsames Constraintstore miteinander interagieren können, soll mit dem in der Abbildung 3.1 dargestellten Beispiel gezeigt werden: Angenommen, sei $S$ ein anfangs leeres Constraintstore und seien $A_1, A_2, A_3, A_4$ die Prozesse, die über S unterschiedliche Aktionen durchführen möchten. Ein Pfeil in das Constraintstore beschreibt eine *tell*-Operation und ein Pfeil aus dem Constraintstore heraus entspricht einer *ask*-Operation. Die Verkettung zweier Aktionen wird mit einem Punkt beschrieben. „$temperatur = 20?.P$“ bedeutet beispielsweise, dass zuerst eine *ask*-Operation für das Constraint $temperatur = 20$ durchgeführt wird. Falls das Constraintstore dieses Constraint im-

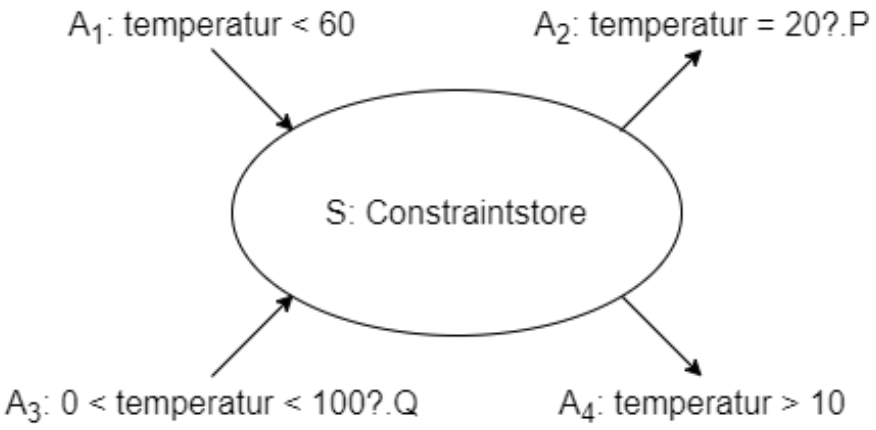


Abbildung 3.1: Interaktion mehrere Prozesse über ein gemeinsames Constraintstore

pliziert, so wird danach der Prozess $P$ ausgeführt (vgl. Wachen oben). Sei außerdem die Reihenfolge der Aktionen so, dass $A_1$ vor $A_2$, $A_2$ vor $A_3$ und $A_3$ vor $A_4$ ausgeführt wird.

Der Prozess $A_1$ fängt an und fügt das Constraint $temperatur < 60$ zum leeren Constraintstore $S$ hinzu. Der Zustand von $S$ wird dann $S = \{temperatur < 60\}$. Der nächste Prozess $A_2$ fragt, ob $temperatur = 20$ aus dem aktuellen Zustand von $S$ hergeleitet werden kann. Das ist jedoch nicht möglich und somit wird der Prozess $P$ auch nicht ausgeführt. Der Prozess $A_2$ wird somit blockiert. Das gleiche gilt auch für Prozess $A_3$, weil das Constraint $0 < temperatur < 100$ auch nicht aus $S = \{temperatur < 60\}$ hergeleitet werden kann. $A_3$ wird damit auch blockiert. $A_4$ ist an der Reihe und versucht, das Constraint $temperatur > 10$ zum Constraintstore hinzufügen. Das Constraint $temperatur > 10$ ist mit S konsistent und kann deswegen zum Constraintstore hinzugefügt werden. Das Constraintstore enthält sodann die beiden Constraints $temperatur < 60$ und $temperatur > 10$. Aus diesem neuen Constraintstore kann aber auch hergeleitet werden, dass das Constraint $0 < temperatur < 100$ von $A_3$ gilt. Damit ist die Bedingung bei $A_3$ erfüllt und der Prozess $Q$ kann auch ausgeführt werden. Es ist natürlich möglich, dass Prozess $Q$ zum Constraintstore weitere Informationen hinzufügt. Falls das Constraintstore von $Q$ so geändert wird, dass das Constraint von Prozess $A_2$ erfüllt werden kann, kann Prozess $P$ auch ausgeführt werden. Sonst bleibt $A_2$ für immer blockiert.

Ein wichtiger Unterschied zum üblicherweise verwendeten Ansatz des gemeinsamen Speichers ist, dass keine Information aus dem Constraintstore gelöscht werden kann [35]. Das heißt, ein Constraintstore wird nur mit Information angereicht, aber es werden daraus nie Informationen entfernt. Dies hat folgende Konsequenzen [35]:

- Die Reihenfolge der Prozessbearbeitung kann andere Prozesse nicht blockieren.
- Hierdurch werden Deadlocks, die nur auf Grund eines ungünstigen Scheduling entstehen, automatisch vermieden.

Durch dieses Modell wird die nebenläufige Programmierung auf eine deklarative Ebene gehoben, weswegen das Modell auch deklarative Nebenläufigkeit (engl. declarative concurrency) genannt wird [35].

Mit Hilfe eines Beispiels kann gezeigt werden, wie diese Konzepte mit den vorhin eingeführten Wachen zusammenarbeiten. Der folgende Prozess hat drei Variablen `X`, `Y`, `Z` und die Aufgabe des Prozesses ist es, dass die dritte Variable immer das Maximum der anderen beiden Variablen annimmt.

```
max(X,Y,Z) := X <= Y | Y = Z
            □ X >= Y | X = Z
```

Mit „:=“ wird ein Prozess eingeleitet. Die Constraints links des senkrechten Strichs „|“ führen eine „ask“-Aktion durch und formulieren somit eine Abfrage. Die Constraints rechts des senkrechten Strichs „|“ sind die „tell“-Constraints, die zum Constraintstore hinzugefügt werden müssen. Die Abfragen links stellen Bedingungen dar, die erfüllt werden müssen, damit „tell“-Aktionen durchgeführt werden können. Mit dem □-Symbol werden Alternativen bezeichnet, die als engagierte Wahl betrachtet werden. Im Gegensatz zur Logikprogrammierung, wo bei mehreren passenden Regeln (für dieses Beispiel wäre dies der Fall für $X = Y$) alle der Reihe nach ausprobiert werden, werden hier beide Alternativen abgearbeitet und der erste Prozess, der mit der Berechnung der Wache fertig ist, darf weiterarbeiten. Der andere Prozess wird terminiert.

Der Prozess $max(X,Y,Z)$ arbeitet dann folgendermaßen:

1. Er wartet, bis entweder $X \leq Y$ oder $X \geq Y$ aus dem Constraintstore gefolgert werden kann (entailment, ask), und
2. Er wählt dann **eine** passende Alternative und fügt ein Gleichheitsconstraint zum Constraintstore hinzu (tell).

Das Modell der nebenläufigen Constraint-Programmierung stellt somit ein sauberes und einfaches Modell für die Synchronisierung nebenläufiger Berechnungen auf der Grundlage von Constraints dar. Für dieses wurden auch mehrere Sprachen entwickelt. AKL war eine der ersten Sprachen, die nebenläufige Constraint-Programmierung umsetzte [45]. Ein weiteres Beispiel ist Oz, für dessen Entwicklung auch ein objekt-orientierter Ansatz verfolgt wurde [40]. Später taten sich die beiden Entwicklerteams zusammen, um Oz und das dazugehörige Programmiersystem Mozart weiterzuentwickeln [29]. Derzeit werden Oz und Mozart in vielen verschiedenen Anwendungsbereichen eingesetzt, in denen die enge Kombination von Nebenläufigkeit und Problemlösungsfähigkeit großes Potential gezeigt hat.

KAPITEL 4

# Sprachen für Constraint-Programmierung

In diesem Kapitel werden drei Sprachen für Constraint-Programmierung vorgestellt. Insbesondere wird die Kernfunktionalität dieser Sprachen diskutiert und es wird darauf eingegangen, welche Domänen von diesen Sprachen unterstützt werden. Außerdem werden die Methoden dieser Sprachen aufgeschlüsselt, um Constraints aus unterschiedlichen Domänen lösen zu können. Bei der Wahl der hier genannten Sprachen wurden einige Kriterien berücksichtigt. Diese sind wie etwa, geschichtliche Bedeutung, Anzahl und Variation der in der Sprache implementierten Constraintdomänen und Neuigkeiten, die durch die Sprache in die Constraint-Programmierung eingeführt wurden.

Die erste Sprache, die genannt wird, ist CHIP, die große historische Bedeutung hat aufgrund der Variation der Constraintdomänen und auch wegen der Konsistenztechniken, die zum ersten Mal in einer CP-Sprache verwendet wurden. Die zweite Sprache ist CLP(R), die ebenfalls eine große historische Bedeutung hat. Diese große historische Bedeutung ist vor allem der Bemühung geschuldet, die Domäne der reellen Zahlen zum ersten Mal in einer CP-Sprache zu unterstützen. Zuletzt werden Constraint Handling Rules vorgestellt. Constraint Handling Rules sind eher als eine Spracherweiterung wahrzunehmen, die dem Benutzer Werkzeuge für die Modellierung neuer Constraintlöser oder für benutzerdefinierte Constraints anbietet.

## 4.1 CHIP

CHIP (Constraint Handling In Prolog) ist eine der ersten Sprachen für Constraint-Programmierung und wurde von Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf und Françoise Berthier im Jahr 1988 am European Computer-Industry Research Centre (ECRC) in München entwickelt [20]. Die Besonderheit von CHIP war die Einführung der endlichen Domänen. Wie der Name impliziert, war CHIP eine Prolog-basierte Sprache [68] und zudem war CHIP die erste CP-Sprache, in der Konsistenztechniken umgesetzt wurden [63].

CHIP bietet dem Programmierer drei Constraintdomänen an. Diese sind:

- Endliche Domänen,
- Boolesche Terme sowie

- Lineare rationale Terme.

Für die endlichen Domänen bietet CHIP nicht nur arithmetische Constraints, sondern auch symbolische, benutzer-definierte Constraints und Constraints höherer Ordnung [20]. Zur Lösung dieser Constraints werden die bereits vorgestellten Konsistenztechniken verwendet. Symbolische Constraints arbeiten wie die zuvor eingeführten globalen Constraints. Constraints dieser Art ermöglichen eine natürliche Problemstellung und eine effizientere Problemlösung [20]. Dabei wird ein bestimmtes Prädikat mit Hilfe von Konsistenztechniken als Constraint behandelt. Die einzige Voraussetzung dafür, dass ein Prädikat als Constraint behandelt werden kann, ist, dass seine Grundinstanzen entweder erfolgreich sind oder in endlicher Zeit scheitern. CHIP ist die erste Sprache, die derartige benutzerdefinierte Constraints erlaubt.

Constraints höherer Ordnung werden zur Modellierung von Optimierungsproblemen verwendet [20]. Ein Beispiel für Constraints dieser Art kann das *minimize*-Constraint sein. Dies hat die Form $minimize(term, function)$ und findet eine Lösung zum gegebenen Term so, dass die ebenfalls gegebene Funktion minimiert wird.

Um Gleichheiten zwischen booleschen Termen zu lösen, werden Boolean-Unifikationsalgorithmen verwendet. Der in CHIP umgesetzte Boolean-Unifikationsalgorithmus ist ein Variablen-Eliminierungs-Algorithmus [20], der den kleinsten gemeinsamen Unifikator ausgibt, falls dieser existiert [47]. Da das Lösen boolescher Constraints ein Entscheidungsverfahren für die Aussagenlogik darstellt und daher NP-vollständig ist, hat jeder Algorithmus für boolesche Constraints exponentielle Worst-Case-Komplexität [28]. Es ist daher sehr wichtig, eine kompakte Beschreibung von booleschen Termen zu verwenden, um eine Lösung effizient berechnen zu können.

Die dritte Domäne, lineare rationale Terme, bezeichnet lineare Terme mit rationalen Zahlen und rationalen Variablen [20]. Die Constraints dieser Domäne sind die arithmetischen Relationen zwischen diesen Termen. CHIP bietet ein Entscheidungsverfahren für Constraints auf linearen rationalen Termen. Das bedeutet, dass CHIP für jede Menge von Constraints (über rationalen Termen) entscheiden kann, ob sie erfüllbar sind oder nicht. Dieses Verfahren ist eine Anpassung des Simplex-Algorithmus, das auf der Eliminierung von Variablen basiert.

Zwei der häufigsten Anwendungsgebiete für CHIP sind Terminplanung und Schaltungsentwurf [20]. Beispielsweise wurde die erste industrielle CP-Anwendung mit der Sprache CHIP für den HIT-Containerhafen (engl. Hongkong International Terminals) in Hongkong entwickelt: Das Ziel war, den Containerschiffen im Hafen Liegeplätze zuzuweisen, und zwar so, dass Ressourcenbedarf wie etwa Stauraum für die Container minimal ist [63].

## 4.2 CLP(R)

Die Sprache CLP(R) wurde als eine Instanziierung des CLP(X)-Schemas an der Monash University, IBM Yorktown Heights und der Carnegie Mellon University entwickelt [44]. Die Grundlagen von CLP(R) wurden im Artikel „The CLP(R) Language and System“ von Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey und Roland Yap vorgestellt [44]. Die Constraintdomäne von CLP(R) ist die lineare arithmetische Domäne über den reellen Zahlen $\mathbb{R}$ (dargestellt durch Fließkommazahlen) [68]. Die Sprache enthält einen vollständigen Constraintlöser für lineare Terme, nicht-lineare Constraints werden dagegen verzögert berechnet.

Das CLP(R)-System besteht aus drei Teilen [47]:

1. Eine Inferenzmaschine (engl. inference engine), die Ableitungsschritte ausführt und Variablenbindungen verwaltet,

2. Ein Modul zur Auswertung komplexer arithmetischer Ausdrücke und zur Umwandlung von Constraints in eine kanonische Form,

3. Ein Constraintlöser, der Constraints löst, die zu kompliziert sind, um direkt in der Inferenzmaschine oder dem arithmetischen Modul behandelt zu werden, und der auch verzögerte (nicht-lineare) Constraints verwaltet.

Der Constraintlöser von CLP(R) enthält wiederum drei Teile: Einen Gleichungslöser, einen Ungleichungslöser und ein „delay"-Modul (dt. Verzögerung) [47]. Der Gleichungslöser verwendet eine Form des Gaußschen Elimination zum Lösen von Gleichungen. Der Ungleichungslöser verwendet eine inkrementelle Form der Simplex-Methode. Nichtlineare Gleichungen werden erst gelöst, sobald sie linear werden, und in diesem Fall an den Gleichungslöser weitergeleitet.

CLP(R) führt eine globale Konsistenzprüfung durch, sodass Widersprüche und Redundanzen in Constraints erkannt werden können [30].

Das vermutlich wichtigste Anwendungsgebiet von CLP(R) ist Elektrotechnik [36]. Die Komplexität der Systeme, die bei elektrotechnischen Problemen entstehen, haben diese Anwendung zu einem Kandidaten für die Benutzung von CLP(R) gemacht. Einige andere Anwendungen für CLP(R) sind Differentialgleichungen, Optionshandel und Musiktheorie [1].

## 4.3 Constraint Handling Rules (CHR)

In klassischen Sprachen für Constraint-Programmierung ist der Constraintlöser fest in die Sprache integriert und kann nicht modifiziert werden [25]. Das heißt, jede Sprache wird mit einem oder mehreren Solvern für bestimmte Arten von Constraints geliefert. Dieser „Blackbox"-Ansatz ist zwar in der Regel sehr effizient, aber unflexibel. Die Definition eines neuen Lösers ist damit oft schwierig: Man muss dessen Implementierung kennen und den Propagierungsalgorithmus implementieren.

Aus dieser Motivation heraus wurden Constraint Handling Rules (CHR) als eine Erweiterung für höhere Programmiersprachen von Thom Frühwirth in „Theory and Practice of Constraint Handling Rules" vorgestellt [25]. CHR ist eine leistungsfähige Sprache zur Modellierung von Lösern, die auf dem Umschreiben von Constraints in einfachere Constraints beruht, bis diese gelöst sind [32]. Mit CHR können benutzerdefinierte Constraints in eine gegebene Host-Sprache eingeführt werden und Verhalten des Constraintlösens kann beeinflusst werden. CHR wird typischerweise als eine Bibliothek bereitgestellt, die auch einen Compiler und ein Laufzeitsystem in der Hostsprache umfasst.

CHR basiert auf zwei Konzepten: Vereinfachung und Propagierung von benutzerdefinierten Constraints [32]. Die Vereinfachungsregeln ersetzen Constraints durch einfachere Constraints, wobei die logische Äquivalenz erhalten bleibt. Propagierungsregeln fügen neue, logisch redundante Constraints hinzu, die zu weiteren Vereinfachungen führen können. Vereinfachungsregeln haben die Form `H⇔G|B` und Propagierungsregeln haben die Form `H⇒G|B`. Dabei besteht `H` aus einer Konjunktion von CHR-Constraints. Die optionale Wache `G` ist eine Konjunktion von eingebauten Constraints. Der Körper `B` ist eine Konjunktion von eingebauten und CHR-Constraints. Ein Beispiel für eine solche Vereinfachungsregel ist `X≤Y∧Y≤X ⇔ X=Y` , während eine mögliche Propagierungsregel `X≤Y∧Y≤Z ⇒ X≤Z` sein kann.

CHR ist nicht unbedingt als eine neue Programmiersprache zu sehen, sondern als eine Spracherweiterung, die in die Syntax mehrerer Wirtssprachen einfügt wird [26]. CHR ist inzwischen in den meisten wichtigen Sprachen für Constraint-Programmierung wie etwa SICStus, SWI oder ECLiPSe implementiert. Darüberhinaus gibt es auch Implementierungen von CHR für viele Allzwecksprachen wie etwa Haskell, C, Java [26].

# KAPITEL 5

# Beispielhafte Verwendungen von Constraint-Programmierung

Constraint-Programmierung hat sich für viele Anwendungsgebiete als nützlich und vorteilhaft erwiesen. Aus Platzgründen werden im Folgenden zwei dieser Anwendungsgebiete mit jeweils einem Beispiel erläutert. Diese Beispiele sind den Anwendungsgebieten automatisierte Planung und Verifikation digitaler Schaltungen entnommen.

Das Potential, das Constraint-Programmierung bei der automatisierten Planung gezeigt hat, war einer der wichtigsten Gründe für die Weiterentwicklung von Constraint-Programmierung. Besonders mit der endlichen Constraintdomäne wurden für die Probleme der automatisierten Planung ein neuer Lösungsansatz vorgeschlagen. Deswegen soll die Lösung eines einfachen aber anschaulichen Problems aus dieser Problemdomäne als Beispiel dienen.

Das zweite Beispiel wird dem Anwendungsgebiet der Verifikation digitaler Schaltungen entnommen, wobei für die Lösung die boolesche Domäne verwendet wird.

## 5.1 Automatisierte Planung

Dieses Beispiel ist inspiriert durch das Planungsbeispiel aus dem Artikel „Constraint Logic Programming - An Informal Introduction“ von Thom Frühwirth et al [28]. Sei im Folgenden eine Fabrik gegeben, in der die Produktherstellung aus den Produktionsschritten $S1, S2, S3, S4, S5, S6$ besteht. Es wird angenommen, dass jeder Schritt eine Stunde dauert und die vollständige Herstellung fünf Stunden dauern soll.

Zwischen den einzelnen Schritten existieren Abhängigkeiten, die in Abbildung 5.1 dargestellt sind. Dabei muss eine Teilaufgabe links eines Striches vor dem Schritt rechts des Striches erledigt werden. Außerdem gibt es die Einschränkung, dass die Schritte $S2$ und $S3$ nicht gleichzeitig erledigt werden können (z.B. weil diese Schritte unter Aufsicht der gleichen Person durchgeführt werden müssen).

Das beschriebene Problem kann als ein Constrainterfüllbarkeitsproblem ausgedrückt werden, indem den Teilaufgaben $Si$, $1 \leq i \leq 6$, Zeitpunkte aus der Menge $\{1,2,3,4,5\}$, die den fünf Stunden entsprechen, zugewiesen werden. Die durch die Abbildung veranschaulichten Constraints können folgendermaßen geschrieben werden:

```
before(S1,S2)
before(S1,S3)
```

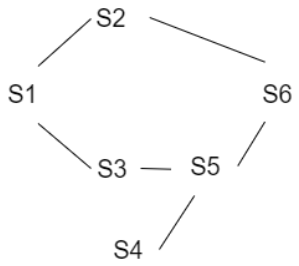


Abbildung 5.1: Abhängigkeiten zwischen einzelnen Schritten

```
before(S2,S6)
before(S3,S5)
before(S4,S5)
before(S5,S6)
notequal(S2,S3)
```

Durch die Constraintpropagierung werden die Domänen der Variablen reduziert. Wird etwa die Bogen-Konsistenz angewandt auf das Constraint `before(S1,S2)`, so ergibt sich für die Variable $S1$ die Wertemenge $\{1,2,3,4\}$ und für die Variable S2 die Wertemenge $\{2,3,4,5\}$. Bogen-Konsistenz wird auf allen `before`-Constraints angewandt, bis die Domänen einzelner Variablen durch Constraintpropagierung nicht mehr reduziert werden können. Es ergeben sich damit folgende Domänen für die Anfangszeitpunkte der Teilaufgaben:

$$S1 \in \{1,2\}, S2 \in \{2,3,4\}, S3 \in \{2,3\}, S4 \in \{1,2,3\}, S5 \in \{3,4\}, S6 \in \{4,5\}$$

Diese Domänen enthalten immer noch inkonsistente Werte, doch diese können durch Konsistenztechniken nicht eliminiert werden. Deswegen muss auch eine Suche durchführt werden, in der Annahmen für Variablen getroffen werden müssen. In dieser Suche wird zum Beispiel die Variable $S1$ mit dem Wert 2 belegt. Diese Annahme erlaubt weitere Propagierung, sodass sich für die Domänen von $S2$ und $S3$ die Wertebereiche $\{3,4\}$ und $\{3\}$ ergeben. Die Propagierung wird fortgesetzt und es entsteht die folgende Lösung für das Problem:

$$S1 \in \{2\}, S2 \in \{4\}, S3 \in \{2\}, S4 \in \{1,2,3\}, S5 \in \{4\}, S6 \in \{5\}$$

Diese muss nicht notwendigerweise die einzige Lösung sein, weil durch die Annahme $S1 = 1$ eine andere Lösung entstehen kann. Jedoch lässt sich diese Lösung wie folgt interpretieren: Damit die Constraints erfüllt sind, müssen die Teilaufgaben $S1, S2, S3, S5, S6$ zu bestimmten Zeitpunkten beginnen aber der Anfangszeitpunkt der Teilaufgabe $S4$ kann relativ frei gewählt werden.

## 5.2 Verifikation Digitaler Schaltungen

Das nächste Beispiel ist dem Anwendungsgebiet der Verifikation digitaler Schaltungen entnommen und entstammt dem Artikel „Practical Applications of Constraint Programming“ von Mark Wallace [77].

Die Idee der Schaltungsverifikation geht auf die folgende Idee zurück [70]: Komplexe Systeme lassen sich auf sehr unterschiedliche Weise zerlegen und modellieren (in funktionale Komponenten, physikalische Komponenten, kausale Abläufe usw.). Für die Verifikation von Schaltungen sollten zwei Modelle, das Verhaltensmodell und das funktionelle

Modell, äquivalent sein. Die Verifikation entspricht dann dem Nachweis der Äquivalenz der beiden Modelle.

Das Verhalten einer Schaltung wird als ein einziges globales Constraint modelliert, das sich aus den Constraints zusammensetzt, die das Verhalten ihrer Komponenten modellieren. Ein Volladdierer in einer digitalen Schaltung realisiert beispielsweise Constraints zwischen seinen Eingängen und Ausgängen, die sich durch die Constraints der Gatter seiner Komponenten wie folgt erfassen lassen:

```
full_adder(XIn, YIn, CarryIn, Sum, CarryOut):-
    and2(XIn, YIn, C1),
    xor(XIn, YIn, S1),
    and2(CarryIn, S1, C2),
    xor(CarryIn, S1, Sum),
    or(C1, C2, CarryOut).
```

Der Volladdierer wird wiederum als Komponente in komplexeren Schaltungen, wie dem n-Bit-Addierer, verwendet. Durch Rekursion kann eine einzige Regel für einen n-Bit-Addierer für einen beliebigen Wert von n wie folgt definiert werden, wobei „[]“ eine leere Liste und „[Head | Tail]“ eine nichtleere Liste bezeichnet:

```
n_bit_adder(XBits, YBits, CarryIn, SumBits, CarryOut):-
    XBits = [], YBits = [] |
        SumBits = [], CarryOut = CarryIn;
    XBits = [X|XT], YBits = [Y|YT], SumBits = [Sum|SumT] |
        full_adder(X, Y, CarryIn, Sum, Carry),
        n_bit_adder(XT, YT, Carry, SumT, CarryOut).
```

Wenn keine weiteren Bits in der Eingabe vorhanden sind, ist die erste Wache erfüllt, wenn jedoch weitere Bits in der Eingabe vorhanden sind, ist die zweite Wache erfüllt.

Digitale Schaltungen werden aus einer Hierarchie von Teilschaltungen aufgebaut, und ihre Constraints werden wiederum aus den mit ihren Teilschaltungen assoziierten Constraints gebildet, wobei in der Regel intentionale Bezeichnungen wie die oben genannte verwendet werden. Letztendlich werden alle Constraints aus den zugrundeliegenden booleschen Funktionen gebildet. Der Boolesche Unifikationsalgorithmus zur Behandlung dieser Klasse von Constraints extrahiert die allgemeinste Lösung für dieses globale Constraint: Die Lösung ist eine Darstellung des Verhaltens der gesamten Schaltung für jede beliebige Eingabe.

Die Verifikation von Schaltungen beruht sodann auf einem Vergleich von Schaltungen in dem Sinne, dass zu Eingangswerten einer Schaltung Ausgangswerte berechnet werden. Es werden jedoch nicht alle Kombinationen von Eingangswerten zur Berechnung herangezogen, sondern nur jene, die für einen gewissen Ausgangswert relevant sind. Diese Menge an Eingangswerten wird mit Hilfe der Constraints ermittelt. Liefern zwei unterschiedliche Schaltungen zu denselben Eingangswerten denselben Ausgangswert, so werden die beiden Schaltungen als äquivalent angesehen und die Verifikation war erfolgreich.

KAPITEL 6

# Fazit

## 6.1 Zusammenfassung

Das Ziel dieser Arbeit war es, einen Überblick über die Constraint-Programmierung zu geben. Dafür wurden zuerst die Grundlagen eingeführt und danach wurden verschiedene Constraint-Domänen und die dabei verwendeten Methoden beispielhaft erläutert. Anschließend wurden nebenläufige und inkrementelle Constraint-Programmierung anhand von Anwendungsbeispielen vorgestellt. Schließlich wurden einige Sprachen für Constraint-Programmierung und zwei Anwendungsbeispiele von Constraint-Programmierung präsentiert.

Constraint-Programmierung erweitert Logikprogrammierung mit neuen Berechnungsdomänen. Durch diese Erweiterung wird die Deklarativität und kurze Entwicklungszeiten der Logikprogrammierung mit der hohen Ausdruckskraft und Anwendungsrelevanz des Constraintlösens kombiniert, weshalb Constraint-Programmierung von einigen als ein Schritt zum heiligen Gral der Programmierung bezeichnet wird [5].

Der Erfolg von Constraint-Programmierung in unterschiedlichen Anwendungsgebieten, insbesondere bei kombinatorischen Suchproblemen, verlangt danach, über dieses Programmierparadigma informiert zu sein. Constraint-Programmierung kann bei vielen Problemen die Perspektive erweitern.

## 6.2 Künftige Forschungsthemen

In diesem letzten Kapitel soll kurz die Zukunft der Constraint-Programmierung erläutert werden.

Eine Gefahr für die Verbreitung von Constraint-Programmierung in der Industrie ist die fehlende Standardisierung [63]. Es existiert keine allgemein akzeptierte Syntax für Constraints. Wie bereits in Kapitel 2.2.2 erwähnt wurde, hat etwa jeder CLP(FD)-Löser seine eigene Syntax zur Definition von Domänen und es ist auch möglich, dass gleiche globale Constraints unter unterschiedlichen Namen bekannt sind.

Das zweite Problem ergibt sich aus imperativen und objektorientierten Sprachen [63]: Viele Löser sind auch mit C++ oder Java-Syntax verfügbar, wodurch die Vorteile der Logikprogrammierung aufgegeben werden, jedoch mit dem Vorteil einer einfacheren Integration

in bereits bestehende Anwendungen.

Sprachen für Constraint-Programmierung sind in der Regel nur durch erfahrene Benutzer anwendbar [63]. Das bedeutet, dass effiziente Suchstrategien und Heuristiken vom naiven Benutzer oft nicht verstanden werden. Verbesserungen bei der Benutzerfreundlichkeit kann die Verbreitung von Constraint-Programmierung erhöhen.

Die Forschung der Constraint-Programmierung konzentriert sich stark auf Effizienzgewinne, wofür Verbesserungen in unterschiedlichen Aspekten notwendig sind. Es wird zum Beispiel erwartet, dass die Entwicklung schnellerer und effizienterer Löser die Akzeptanz von Constraint-Programmierung allgemein verbessern und die Aufnahme von Techniken, Konstrukten und Bibliotheken der Constraint-Programmierung in konventionelle Sprachen fördern wird [75].

Eine weitere Steigerung der Effizienz ist beim Suchverhalten möglich, wo vorgeschlagen wird, das chronologische Backtracking-Suchmodell von CP-Sprachen mit lokalen Suchtechniken zu kombinieren [55].

Ein weiterer Ansatz zur Erhöhung der Effizienz besteht in Parallelisierung [55]. Constraint-Programmierung ist aufgrund seiner Deklarativität zur Parallelisierung gut geeignet. Es wird vermutet, dass die kommende Hardware-Generation für weitere Geschwindigkeitssteigerungen auf Parallelität angewiesen sein wird und dass deklarative Paradigmen wie Constraint-Programmierung daher immer wichtiger werden könnten.

Mit dem Aufkommen von Big Data sind noch zwei Herausforderungen für Constraint-Programmierung erschienen: Skalierbarkeit und Unbeständigkeit (engl. uncertainty) [23]. Das exponentiell wachsende Datenvolumen in allen Anwendungsbereichen erfordert, dass sich auch Constraint-Programmierung an diese neue Bedingung anpasst. Unbeständigkeit bezieht sich auf die dynamische Natur von realen Anwendungen. Beispielsweise kann bei Terminplanung die Dauer einer Aufgabe ungewiss sein, während bei Bestandsverwaltung (engl. inventory management) die Kundennachfrage ungewiss sein kann.

Ein weiteres Forschungsgebiet sind Constraint-Datenbanken. Die Grundidee von Constraint-Datenbanken besteht darin, den Begriff des Tupels in einer relationalen Datenbank zu einer Konjunktion von Constraints zu verallgemeinern [48]. Den Grundstein für diesen Bereich legten in „Constraint Query Languages“ Paris C. Kanellakis, Gabriel M. Kuper und Peter Z. Revesz [46].

Ein ebenso vielversprechender Forschungsschwerpunkt ist Constraint-Lernen. Der Begriff Constraint-Lernen bezieht sich auf das Problem, eine Menge von Constraints zu finden, die einen gegebenen Datensatz erfüllen [61]. Constraint-Lernen wird verwendet, wenn die Struktur oder die Parameter des zu lösenden Constrainterfüllbarkeits- oder Constraintsoptimierungsproblems (teilweise) unbekannt sind und aus Beispielen abgeleitet werden müssen [72].

# Literaturverzeichnis


[1] *The CLP(R) Programmer's Manual: Version 1.1*.

[2] Akira Aiba, Kô Sakai, Yosuke Sato, David J. Hawley, and Ryuzo Hasegawa, *Constraints Logic Programming Language CAL*, FGCS, 1988.

[3] Krzysztof Apt, *Principles of constraint programming*, Cambridge University Press, Cambridge, 2003.

[4] Krzysztof Apt and Mark Wallace, *Constraint logic programming using ECLiPSe*, January 2007, Journal Abbreviation: Constraint Logic Programming Using Eclipse Publication Title: Constraint Logic Programming Using Eclipse.

[5] Roman Barták, *Constraint programming: In pursuit of the holy grail*, Proceedings of WDS99 (invited lecture) (1999).

[6] S. Baselice, P. A. Bonatti, and M. Gelfond, *Towards an Integration of Answer Set and Constraint Solving*, Logic Programming (Berlin, Heidelberg) (Maurizio Gabbrielli and Gopal Gupta, eds.), Lecture Notes in Computer Science, Springer, 2005, pp. 52–66 (en).

[7] Nicolas Beldiceanu, Mats Carlsson, and jean-xavier Rampon, *Global Constraint Catalog*, May 2005.

[8] ———, *Global Constraint Catalog*, Technical Report SICS Technical Report T2005:08, May 2005.

[9] Frédéric Benhamou, William Older, and Andre Vellino, *Constraint Logic Programming on Boolean, Integer and Real Intervals*, (2001).

[10] Alexander Bockmayr and John N. Hooker, *Constraint Programming*, Handbooks in Operations Research and Management Science (K. Aardal, G. L. Nemhauser, and R. Weismantel, eds.), Discrete Optimization, vol. 12, Elsevier, January 2005, pp. 559–600 (en).

[11] Alan Borning, *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, ACM Transactions on Programming Languages and Systems **3** (1981), no. 4, 353–387.

[12] Alan Hamilton Borning, *Thinglab–a constraint-oriented simulation laboratory.*, phd, Stanford University, Stanford, CA, USA, 1979, AAI7917213.

[13] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei, *Foundations of Rule-Based Query Answering*, Reasoning Web, Third International Summer School 2007, LNCS, vol. 4636, Springer-Verlag, 2007.

[14] Jacques Cohen, *Constraint logic programming languages*, Communications of the ACM **33** (1990), no. 7, 52–68 (en).

[15] A. Colmerauer, *Prolog II Reference Manual and Theoretical Model*, Tech. report, U Aix-Marseille, GroupeIA, October 1982.

[16] Alain Colmerauer, *An introduction to Prolog III*, Communications of the ACM **33** (1990), no. 7, 69–90 (en).

[17] Alain Colmerauer and Philippe Roussel, *The birth of Prolog*, History of programming languages—II, Association for Computing Machinery, New York, NY, USA, January 1996, pp. 331–367.

[18] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck, *Solving large combinatorial problems in logic programming*, The Journal of Logic Programming **8** (1990), no. 1, 75–93 (en).

[19] _____, *Solving a cutting-stock problem with the constraint logic programming language CHIP*, Mathematical and Computer Modelling **16** (1992), 95–105.

[20] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier, *The Constraint Logic Programming language CHIP*, December 1988, pp. 693–702.

[21] Agostino Dovier, Andrea Formisano, and Enrico Pontelli, *An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems*, Journal of Experimental & Theoretical Artificial Intelligence **21** (2009), 79–121.

[22] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning, *An incremental constraint solver*, Communications of the ACM **33** (1990), no. 1, 54–63.

[23] Eugene C. Freuder and Barry O'Sullivan, *Grand challenges for constraint programming*, Constraints **19** (2014), no. 2, 150–162 (en).

[24] T. Fruhwirth and S. Abdennadher, *Principles of constrain systems and constraint solvers*, Archives of Control Sciences **16** (2006), 131 – 159 (en).

[25] Thom Frühwirth, *Theory and practice of constraint handling rules*, The Journal of Logic Programming **37** (1998), no. 1, 95–138 (en).

[26] _____, *Welcome to Constraint Handling Rules*, Constraint Handling Rules (Tom Schrijvers and Thom Frühwirth, eds.), vol. 5388, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, Series Title: Lecture Notes in Computer Science, pp. 1–15 (en).

[27] Thom Frühwirth and Slim Abdennadher, *Essentials of Constraint Programming*, Springer Berlin Heidelberg, 2003.

[28] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace, *Constraint Logic Programming - An Informal Introduction*, Logic Programming in Action (Berlin, Heidelberg) (G. Comyn, N. E. Fuchs, and M. J. Ratcliffe, eds.), Springer Berlin Heidelberg, 1992, pp. 3–35.

[29] Thom Frühwirth, Laurent Michel, and Christian Schulte, *Constraints in Procedural and Concurrent Languages*, Handbook of Constraint Programming, Elsevier, 2006, pp. 451–492.

[30] Thomas Frühwirth, *Constraint Logic Programming - An Overview*, Tech. report, Christian Doppler Laboratory für Expert System, Vienna, Austria, August 1990.

[31] Hervé Gallaire, *Logic programming: Further developments*, IEEE Symposium on Logic Programming (Boston), 1985, pp. 88–99.

[32] Marco Gavanelli and Francesca Rossi, *Constraint Logic Programming*, A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming, GULP (Agostino Dovier and Enrico Pontelli, eds.), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pp. 64–86 (en).

[33] Jürgen Giesl, *Logikprogrammierung*, 2020.

[34] The Constraint Programming Working Group, *Constraint Programming*, 1996.

[35] Michael Hanus, *Prinzipien von Programmiersprachen*, 2020.

[36] Nevin Heintze, Spiro Michaylov, and Peter Stuckey, *CLP(R) and some electrical engineering problems*, Journal of Automated Reasoning **9** (1992), no. 2, 231–260 (en).

[37] Pascal Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[38] _____, *Incremental Constraint Satisfaction in Logic Programming*, Logic programming, MIT Press, Cambridge, MA, USA, May 1990, pp. 189–202.

[39] Pascal Van Hentenryck, H. Simonis, and Mehmet Dincbas, *Constraint Satisfaction Using Constraint Logic Programming*, Artif. Intell. (1992).

[40] Martin Henz, *Objects For Concurrent Constraint Programming*, THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE, SPRINGER SCIENCE+BUSINESS MEDIA. LLC, 1998.

[41] John Hooker, *Logic, Optimization, and Constraint Programming*, INFORMS Journal on Computing **14** (2002), 295–321.

[42] J. Jaffar and J.-L. Lassez, *Constraint Logic Programming*, Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (New York, NY, USA), POPL '87, Association for Computing Machinery, 1987, pp. 111–119.

[43] Joxan Jaffar and Michael J. Maher, *Constraint logic programming: a survey*, The Journal of Logic Programming **19-20** (1994), 503–581 (en).

[44] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap, *The CLP(R) language and system*, ACM Transactions on Programming Languages and Systems **14** (1992), no. 3, 339–395.

[45] Sverker Janson, *AKL - A Multiparadigm Programming Language*, PhD, Uppsala University, SICS, Kista, Sweden, 1994.

[46] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz, *Constraint Query Languages*, Tech. report, IBM, Brown University, 1990.

[47] Frank Kriwaczek, *An introduction to constraint logic programming*, Advanced Topics in Artificial Intelligence (Berlin, Heidelberg) (Vladimír Mařík, Olga Štěpánková, and Rorbert Trappl, eds.), Lecture Notes in Computer Science, Springer, 1992, pp. 82–94 (en).

[48] Gabriel Kuper, Leonid Libkin, and Jan Paredaens, *Introduction*, Constraint Databases (Gabriel Kuper, Leonid Libkin, and Jan Paredaens, eds.), Springer, Berlin, Heidelberg, 2000, pp. 1–16 (en).

[49] Catherine Lassez, Ken McAloon, and Roland Yap, *Constraint Logic Programming and Option Trading*, IEEE Expert **2** (1987), no. 3, 42–50, Conference Name: IEEE Expert.

[50] Michael Leuschel, *Logic Program Specialisation*, Partial Evaluation (Berlin, Heidelberg) (John Hatcliff, Torben Mogensen, and Peter Thiemann, eds.), Lecture Notes in Computer Science, Springer, 1999, pp. 155–188 (en).

[51] John Wylie Lloyd, *Foundations of Logic Programming*, 1 ed., Symbolic Computation, no. 1064, Springer Berlin, Heidelberg, 1984 (en).

[52] F. Javier López-Fraguas, Mario Rodríguez-Artalejo, and Rafael del Vado Vírseda, *Constraint Functional Logic Programming Revisited*, Electronic Notes in Theoretical Computer Science **117** (2005), 5–50 (en).

[53] Toni Mancini, Davide Micaletto, Fabio Patrizi, and Marco Cadoli, *Evaluating ASP and Commercial Solvers on the CSPLib*, Constraints **13** (2008), no. 4, 407–436 (en).

[54] Kim Marriott and Peter J. Stuckey, *Programming with Constraints - An Introduction*, The MIT Press, 1998 (English).

[55] Kim Marriott, Peter J. Stuckey, and Mark Wallace, *Constraint Logic Programming*, Handbook of Constraint Programming, Elsevier, 2006, pp. 407–450.

[56] Francisco Menezes, Pedro Barahona, and Codognet Philippe, *An Incremental Hierarchical Constraint Solver.*, January 1993, pp. 190–199.

[57] Ian Miguel and Patrick Prosser, *What problems will AI solve in future? An old British gameshow can help explain*.

[58] Gregory M. Oster and Anthony J. Kusalik, *Icola — Incremental Constraint-Based Graphics for Visualization*, Constraints **3** (1998), no. 1, 33–59 (en).

[59] Barth Peter, *Constraint Logic Programming*, Logic-Based 0–1 Constraint Programming, Operations Research/Computer Science Interfaces Series, vol. 5, Springer, Boston, MA, 1996, pp. 9 – 44.

[60] Frank Pfenning, *Logic Programming*, 2007.

[61] L. D. Raedt, Andrea Passerini, and Stefano Teso, *Learning Constraints From Examples*, AAAI, 2018.

[62] Edited F Rossi, P van Beek, and T Walsh (eds.), *Handbook of Constraint Programming*, Elsevier Science Inc., 2006 (en).

[63] Francesca Rossi, *Constraint (Logic) Programming: A Survey on Research and Applications*, New Trends in Constraints (Berlin, Heidelberg) (Krzysztof R. Apt, Eric Monfroy, Antonis C. Kakas, and Francesca Rossi, eds.), Lecture Notes in Computer Science, Springer, 2000, pp. 40–74 (en).

[64] Vijay Saraswat, Martin Rinard, and Prakash Panangaden, *Semantic Foundations of Concurrent Constraint Programming.*, January 1991, pp. 333–352.

[65] Vijay A. Saraswat, *Concurrent Constraint Programming*, MIT Press, 1993.

[66] Vijay A. Saraswat and Martin Rinard, *Concurrent constraint programming*, Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '90, Association for Computing Machinery, December 1989, pp. 232–245.

[67] H. Simonis, *Constraint Logic Programming as a Digital Circuit Design Tool*, Ph.D. thesis, 1992.

[68] Helmut Simonis, *Constraint Logic Programming*, - (2008), 37.

[69] Guy Lewis Steele, *The Definition and Implementation of a Computer Programming Language Based on Constraints*, Ph.D. thesis, M.I.T., August 1980, Accepted: 2004-10-20T20:08:23Z.

[70] Gerald Jay Sussman and Guy Lewis Steele, *Constraints—A language for expressing almost-hierarchical descriptions*, Artificial Intelligence **14** (1980), no. 1, 1–39 (en).

[71] Ivan Edward Sutherland, *Sketchpad: A man-machine graphical communication system*, PhD, Massachusetts Institute of Technology, Cambridge, MA, USA, January 1963.

[72] Stefano Teso, *Constraint Learning: An Appetizer*, Reasoning Web, 2019.

[73] Frank D. Valencia, *Concurrency, Time, and Constraints*, Logic Programming (Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Catuscia Palamidessi, eds.), vol. 2916, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, Series Title: Lecture Notes in Computer Science, pp. 72–101 (en).

[74] Pascal Van Hentenryck and Thierry Le Provost, *Incremental search in Constraint Logic Programming*, New Generation Computing **9** (1991), no. 3, 257 (en).

[75] Pascal Van Hentenryck and Vijay Saraswat, *Strategic directions in constraint programming*, ACM Computing Surveys **28** (1996), no. 4, 701–726.

[76] P. Voda, *The Constraint Language Trilogy: Semantics and Computations*, Technical Report, CLS, Vancouver, Canada, 1988.

[77] Mark Wallace, *Practical applications of constraint programming*, Constraints **1** (1996), no. 1, 139–168 (en).

[78] ———, *Constraint Logic Programming*, Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part I (Antonis C. Kakas and Fariba Sadri, eds.), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2002, pp. 512–532 (en).

[79] Toby Walsh, *Stochastic Constraint Programming*, arXiv:0903.1152 [cs] (2009), 5, arXiv: 0903.1152.