Principles of Constraint Programming

Krzysztof Apt

CAMBRIDGE

CAMBRIDGE

more information - www.cambridge.org/9780521825832

This page intentionally left blank

Principles of Constraint Programming

Krzysztof R. Apt

CWI, Amsterdam, The Netherlands

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, United Kingdom

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org

Information on this title: www.cambridge.org/9780521825832

First published in print format 2003

ISBN-13 978-0-511-05616-1 eBook (Adobe Reader)
ISBN-10 0-511-05616-8 eBook (Adobe Reader)

ISBN-13 978-0-521-82583-2 hardback
ISBN-10 0-521-82583-0 hardback

To Alma and Daniel

Contents

Acknowledgements

I took to constraint programming late. In fact, I can be more precise. In the summer of 1995, when travelling to a conference, I stopped by at Joxan and Jennifer Jaffar's house in upstate New York. Joxan was working then at the IBM Research Heights and they were moving to Singapore. I still recall the half packed boxes in their hall and a pleasant evening spent on their terrace. I was discussing with Joxan research on constraint logic programming and was confused at that time that constraint programming is the same as constraint logic programming. But he showed me the book Tsang [1993]. Most of this material was new to me, though some isolated concepts I encountered earlier in the most useful book Van Hentenryck [1989].

One year later I thought I should give a course on the subject to learn it better. Providentially, in December 1996, during my visit to Israel, Rachel Ben Eliyahu from the University of Beer-sheva 'sent me over' to Rina Dechter who was staying then on sabbatical at the Tel-Aviv University. A two hour discussion with Rina helped me enormously. Moreover, over the next few months Rina patiently replied to all my, often stupid, questions and provided me with most useful pointers to the literature. This helped me to enter this subject and to start doing research on it.

In the course of the next five years I learned a lot on constraint programming by talking to many colleagues. In addition to those mentioned above I recall in particular illuminating conversations on this subject with Maarten van Emden (during his visits at CWI and my visit at the University of Victoria, Canada, in the summer of 1999), with Andrea Schaerf (during his visits at CWI in the period 1996–1998), François Fages (during the POPL '97 conference in Paris), with, unfortunately deceased in 2000, Igor Shetsov (in the bus to the hotel during the CP '97 conference and later, in 2000, in Amsterdam), François Puget (during a flight from Linz to Frankfurt after the CP '97 conference), Philippe Codognet (during his two short visits in

Amsterdam in, I believe, 1996 and 1999), Alexander Semenov (during his visit to Amsterdam in 2001), and last, but not least, Eric Monfroy, who worked at CWI in Amsterdam from 1996 until 2000.

In 2003 the circle closed, since I finished this book while visiting the National University of Singapore, where Joxan is now Dean of the School of Computing. While in Singapore, I also profited from useful conversations on constraint programming with Martin Henz and Roland Yap. Martin kindly provided me with the tests summarised in Table 9.1 in Chapter 9, obtained using the Oz Explorer. I would like to thank here all those mentioned above for their help in my understanding of the subject.

In addition, I appreciated most helpful comments, corrections and suggestions on various parts of the manuscript sent to me by Sebastian Brand, Maarten van Emden, Pierre Flener, Rosella Gennari, Martin Henz, Willem Jan van Hoeve, Eric Monfroy, Zsofi Ruttkay, Kees Vermeulen and Peter Zoeteweij. Also, I have greatly benefited from giving a course based on this material at the School of Computing of the National University of Singapore. Several students in my course, namely Lim Min Kwang, Cui Hang, Wen Conghua, Dong Xiaoan, Ma Lin, Xu Cheng and Damith Chatura Rajapakse provided useful comments. I also taught on this subject for a couple of years at the University of Amsterdam. Strangely, the only comment I ever got on the continuously growing manuscript was from a student who found that I should replace somewhere in the text 'We now get' by 'We get now'. [1]

Finally, Liu Jiang Hong kindly prepared the figures using the Dia drawing program. Nicola Vitacolonna modified the `skak` package in a way that allowed me to take care of the chess board drawings in Chapters 8 and 9 and Hugh Anderson helped me to insert them in the text as .epsi files. Also it is a pleasure to acknowledge the help I received from David Tranah of the Cambridge University Press throughout the production of this book.

And last, but not least, I would like to thank my family, Ruth, Alma and Daniel, for their patience with me during writing this book. Alma (now 7) kindly offered to proofread the book but fortunately it's not needed: this book no errors. Daniel (now 10) got interested in the Dia program. Who knows: perhaps he will be willing to make drawings for my next book?

PS. A comment about the cover: Alma likes penguins very much.

[1] Fortunately, worse comments can happen. Peter Medawar, a recipient of the Nobel Prize in Physiology or Medicine in 1960 mentions in his memoirs that he gave once a lecture during an open day at his University and the only question he got was 'Where is the women's toilet?'

1

Introduction

1.1 Basic characteristics of constraint programming

THIS BOOK IS about ***constraint programming***, an alternative approach to programming which relies on a combination of techniques that deal with ***reasoning*** and ***computing***. It has been successfully applied in a number of fields including molecular biology, electrical engineering, operations research and numerical analysis. The central notion is that of a constraint. Informally, a ***constraint*** on a sequence of variables is a relation on their domains. It can be viewed as a requirement that states which combinations of values from the variable domains are admitted. In turn, a ***constraint satisfaction problem*** consists of a finite set of constraints, each on a subsequence of a given sequence of variables.

To solve a given problem by means of constraint programming we first formulate it as a constraint satisfaction problem. To this end we

- introduce some variables ranging over specific domains and constraints over these variables;
- choose some language in which the constraints are expressed (usually a small subset of first-order logic).

This part of the problem solving is called ***modeling***. In general, more than

one representation of a problem as a constraint satisfaction problem exists. Then to solve the chosen representation we use either

- domain specific methods,

or

- general methods,

or a combination of both.

The ***domain specific methods*** are usually provided in the form of implementations special purpose algorithms. Typical examples are:

- a program that solves systems of linear equations,
- a package for linear programming,
- an implementation of the unification algorithm, a cornerstone of automated theorem proving.

In turn, the ***general methods*** are concerned with the ways of reducing the search space and with specific ***search methods***. The algorithms that deal with the search space reduction are usually called ***constraint propagation algorithms***, though several other names have been often used. These algorithms maintain equivalence while simplifying the considered problem. They achieve various forms of ***local consistency*** that attempt to approximate the notion of (global) consistency. The (top down) search methods combine various forms of constraint propagation with the customary backtrack and branch and bound search.

The definition of constraint programming is so general that it embodies such diverse areas as Linear Algebra, Global Optimization, Linear and Integer Programming, etc. Therefore we should stress one essential point. If domain specific methods are available they should be applied *instead* of the general methods. For example, when dealing with systems of linear equations, the well-known linear algebra algorithms are readily available and it does not make sense to apply to these equations the general methods.

In fact, one of the aims of constraint programming is to look for efficient domain specific methods that can be used instead of the general methods and to incorporate them in a seamless way into a general framework. Such a framework usually supports

- domain specific methods by means of specialised packages, often called ***constraint solvers***,
- general methods by means of various built-ins that in particular ensure or facilitate the use of the appropriate constraint propagation algorithms and support various search methods.

Once we represent a problem as a constraint satisfaction problem we need to solve it. In practice we are interested in:

- determining whether the chosen representation has a solution (is consistent),
- finding a solution, respectively, all solutions,
- finding an optimal solution, respectively, all optimal solutions w.r.t. some quality measure.

After this short preview we can formulate the following basic characteristics of constraint programming:

Two Phases Approach: The programming process consists of two phases: a generation of a problem representation by means of constraints and a solution of it. In practice, both phases consist of several smaller steps that can be interleaved.

Flexibility: The representation of a problem by means of constraints is very flexible because the constraints can be added, removed or modified. This flexibility is inherited by constraint programming.

Presence of Built-ins: To support this approach to programming several built-in methods are available. They deal with specific constraint solvers, constraint propagation algorithms and search methods.

An additional aspect brought in by constraint programming is that modeling by means of constraints leads to a representation of a problem by means of relations. This bears some resemblance to database systems, for instance relational databases. In fact, constraints are also studied in the context of database systems. They are useful in situations where some information, for instance the definition of a region of a map, needs to be provided implicitly, by means of constraints on reals.

The difference is that in the context of database systems the task consists of efficiently querying the considered relations, independently on whether they are defined explicitly (for instance by means of tables) or implicitly (for example by means of recursion or inequalities). In contrast, in constraint programming the considered relations are usually defined implicitly and the task consists of solving them or determining that no solution exists. This leads to different methods and different techniques.

1.2 Applications of constraint programming

Problems that can be best solved by means of constraint programming are usually those that can be naturally formulated in terms of requirements,

general properties, or laws, and for which domain specific methods lead to overly complex formalisations. Constraint programming has already been successfully applied in numerous domains including:

- interactive graphic systems (to express geometric coherence in the case of scene analysis),
- operations research problems (various optimization problems, in particular scheduling problems),
- molecular biology (DNA sequencing, construction of 3D models of proteins),
- business applications (option trading),
- electrical engineering (location of faults in the circuits, computing the circuit layouts, testing and verification of the design),
- numerical computation (solving polynomial constraints with guaranteed precision),
- natural language processing (construction of efficient parsers),
- computer algebra (solving and/or simplifying equations over various algebraic structures).

More recent applications of constraints involve generation of coherent music radio programs, software engineering applications (design recovery and code optimization) and selection and scheduling of observations performed by satellites. Also, constraint programming proved itself a viable approach to tackle certain computationally intractable problems.

While an account of most of these applications cannot be fit into an introductory book, like this one, an interested reader can easily study the research papers on the above topics, after having acquainted himself/herself with the methods explained in this book.

The growing importance of this area can be witnessed by the fact that there are now annual conferences and workshops on constraint programming and its applications that consistently attract more than one hundred (occasionally two hundred) participants. Further, in 1996 an (unfortunately expensive) journal called 'Constraints' was launched. Also, several special issues of computer science journals devoted to the subject of constraints have appeared. But the field is still young and only a couple of books on this subject have appeared so far. This led us to writing this book.

1.3 A very short history of the subject

Before we engage in our presentation of constraint programming, let us briefly summarise the history of this subject. It will allow us to better understand the direction the field is heading.

The concept of a constraint was used already in 1963 in an early work of I. Sutherland on an interactive drawing system SKETCHPAD. In the seventies various experimental languages were proposed that used the notion of constraints and relied on the concept of constraint solving.

The concept of a constraint satisfaction problem was also formulated in the seventies by researchers in the artificial intelligence (AI). They also identified the main notions of local consistency and the algorithms that allow us to achieve them. Independently, various search methods were defined. Some of them, like backtracking can be traced back to the nineteenth century, while others, like branch and bound, were defined in the context of combinatorial optimization. The contribution of constraint programming was to identify various new forms of search that combine the known techniques with various constraint propagation algorithms. Some specific combinations were already studied in the area of combinatorial optimization.

In the eighties the first constraint programming languages of importance were proposed and implemented. The most significant were the languages based on the logic programming paradigm. This led to a development of ***constraint logic programming***, an extension of logic programming by the notion of constraints. The programming view that emerged led to an identification of ***constraint store*** as a central concept. Constraint propagation and various forms of search are usually available in these languages in the form of built-ins.

In the late eighties and the nineties a form of synthesis between these two developments took place. The researchers found various new applications of constraint programming, most notably in the fields of operations research and numerical analysis. The progress was often achieved by identifying important new types of constraints and new constraint propagation algorithms. One also realised that further progress may depend on a combination of techniques from AI, operations research, computer algebra and mathematical logic. This turned constraint programming into an interesting hybrid area, in which theoretical work is often driven by applications and in turn applications lead to new challenges concerning implementations of constraint programming.

1.4 Our approach

In our presentation of the basic concepts and techniques of constraint programming we strive at a streamlined presentation in which we clarify the nature of these techniques and their interrelationship. To this end we organised the presentation around a number of simple principles.

Principle 1: Constraint programming is about a formulation of the problem as a constraint satisfaction problem and about solving it by means of domain specific or general methods.

This explains our focus on the constraint satisfaction problems and constraint solvers.

Principle 2: Many constraint solvers can be naturally explained using a rule-based framework. The constraint solver consists then of a set of rules that specify its behaviour and a scheduler. This viewpoint stresses the connections between rule-based programming and constraint programming.

This explain our decision to specify the constraint solvers by means of proof rules that transform constraint satisfaction problems.

Principle 3: The constraint propagation algorithms can be naturally explained as instances of simple generic iteration algorithms.

This view allows us to clarify the nature of the constraint propagation algorithms. Also, it provides us with a natural method for implementing the discussed constraint solvers, since a rule scheduler is just another instance of a generic iteration algorithm.

Principle 4: (Top down) search techniques can be conceptually viewed as traversal algorithms of the search trees.

This explains why we organised the chapter on search around the slogan:

Search Algorithm = Search Tree + Traversal Algorithm,

and why we explained the resulting algorithms in the form of successive reformulations.

1.5 Organisation of the book

The above explained principles lead to a natural organisation of the material. Here is a short preview of the remaining chapters. In **Chapter 2** we discuss several examples of constraint satisfaction problems. We stress there that in many situations several natural representations are possible. In **Chapter 3** we introduce a general framework that allows us to explain the basics of constraints programming. We identify there natural ingredients of this

framework. This makes it easier to understand the subject of the subsequent chapters.

Then, in **Chapter 4**, we provide three well-known examples of complete constraint solvers. They deal, respectively, with solving equations over terms, linear equations over reals and linear inequalities over reals. In turn, in **Chapter 5** we introduce several notions of local consistency and characterise them in the form of proof rules. These notions allow us to study in **Chapter 6** in more detail a number of incomplete constraint solvers that involve Boolean constraints and linear and arithmetic constraints on integers and reals.

In **Chapter 7** we study the constraint propagation algorithms that allow us achieve the forms of local consistency discussed in Chapter 5. The characterisation of these notions in the form of proof rules allows us to provide a uniform presentation of these algorithms as instances of simple generic iteration algorithms. Next, in **Chapter 8**, we discuss various (top down) search algorithms. We present them in such a way that one can see how these algorithms are related to each other. Finally, in **Chapter 9**, we provide a short overview of the research directions in constraint programming.

Those interested in using this book for teaching may find it helpful to use the transparencies that can be downloaded from the following website: `http://www.cwi.nl/~apt/pcp`.

2

Constraint satisfaction problems: examples

THE AIM OF this chapter is to discuss various examples of constraint satisfaction problems (CSPs [2] in short). The notion of a CSP is very general, so it is not surprising that these examples cover a wide range of topics. We limit ourselves here to the examples of CSPs that are simple to explain and that illustrate the use of general methods of constraint programming. In particular, we included here some perennial puzzles, since, as it has been recognised for some time, they form an excellent vehicle to explain certain principles of constraint programming.

As already mentioned in Chapter 1 the representation of a problem as a CSP is usually called ***modeling***. The selected examples clarify a number of aspects of modeling. First, as we shall see, some of the problems can be formalised as a CSP in a straightforward way. For other problems the appropriate representation as a CSP is by no means straightforward and relies on a non-trivial 'background' theory that ensures correctness of the

[2] For those knowledgeable in other areas of computer science: constraint satisfaction problems have nothing do to with Communicating Sequential Processes, a programming notation for distributed processes introduced by C.A.R. Hoare and also abbreviated to CSP.

adopted representation. Also for several problems, more than one natural representation exists.

When presenting the CSPs it is useful to classify them according to some criterion. In general, the techniques used to solve CSPs depend both on the domains over which they are defined and on the syntax of the used constraints. In most examples we use some simple language to define the constraints. Later, in Chapters 4 and 6, we shall be more precise and shall discuss in detail specific languages in which the constraints will be defined. But now it is too early to appreciate the role played by the syntax. So we rather classify the CSPs according to the domains over which they are defined. This explains the structure of this chapter.

First, we formalise in Section 2.1 the notion of a constraint and of a CSP. Then, in Section 2.2 we introduce some well-known problems and puzzles that can be naturally formalised as CSPs with integer domains. In Section 2.3 we consider examples of problems the formalisation of which leads to CSPs with variables ranging over reals. In turn, in Section 2.4 we consider ***Boolean CSPs***. These are CSPs in which the variables range over the integer domain [0..1] or, equivalently, {**false, true**} and in which the constraints are expressed by means of Boolean expressions.

An important class of CSPs are the ones in which the variables range over non-numeric domains. We call them ***symbolic CSPs***. They are considered in Section 2.5. In case we are interested in finding an optimal solution to a CSP we associate with each solution an objective function that we want to minimise or maximise. This leads to a modification of a CSP that we call a ***constrained optimization problem***. They are considered in Section 2.6.

2.1 Basic concepts

As explained in the previous chapter constraint satisfaction problems, or CSPs, are a fundamental concept in constraint programming. To proceed we need to define them formally. The precise definition is completely straightforward. First we introduce the notion of a constraint.

Consider a finite sequence of variables $Y := y_1, \ldots, y_k$ where $k > 0$, with respective domains $D_1, \ldots, D_k$ associated with them. So each variable y_i ranges over the domain D_i. By a ***constraint*** C on Y we mean a subset of $D_1 \times \cdots \times D_k$. When $k = 1$ we say that the constraint is ***unary*** and when $k = 2$ that the constraint is ***binary***. By a ***constraint satisfaction problem***, or a ***CSP***, we mean a finite sequence of variables $X := x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write such a CSP as $\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$, where

$\mathcal{DE} := x_1 \in D_1, \ldots, x_n \in D_n$ and call each construct of the form $x \in D$ a ***domain expression***. To simplify the notation we omit the '{ }' brackets when presenting specific sets of constraints $\mathcal{C}$.

We now define the crucial notion of a solution to a CSP. Intuitively, a solution to a CSP is a sequence of legal values for all of its variables such that all its constraints are satisfied. More precisely, consider a CSP $\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$ with $\mathcal{DE} := x_1 \in D_1, \ldots, x_n \in D_n$. We say that an n-tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ ***satisfies*** a constraint $C \in \mathcal{C}$ on the variables $x_{i_1}, \ldots, x_{i_m}$ if

$$(d_{i_1}, \ldots, d_{i_m}) \in C.$$

Then we say that an n-tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ is a ***solution*** to $\langle \mathcal{C} \ ; \ \mathcal{DE} \rangle$ if it satisfies every constraint $C \in \mathcal{C}$. If a CSP has a solution, we say that it is ***consistent*** and otherwise we say that it is ***inconsistent***.

Note that in the definition of a constraint and of a CSP no syntax was assumed. In practice, of course, one needs to define the constraints and the domain expressions. In what follows we assume that they are defined in some specific, further unspecified, language. In this representation it is implicit that each constraint is a subset of the Cartesian product of the associated variable domains. For example, if we consider the CSP $\langle x < y \ ; \ x \in [0..10], y \in [5..10] \rangle$, then we view the constraint $x < y$ as the set of all pairs (a, b) with $a \in [0..10]$ and $b \in [5..10]$ such that $a < b$.

Let us illustrate these concepts by a simple example. Consider the sequence of four variables x, y, z, u ranging over natural numbers and the following three constraints on them: $x^3 + y^3 + z^3 + u^3 = 100$, $x < u$, $x + y = z$. According to the above notation we write this CSP as

$$\langle x^3 + y^3 + z^3 + u^3 = 100, \ x < u, \ x + y = z \ ; \ x \in \mathcal{N}, y \in \mathcal{N}, z \in \mathcal{N}, u \in \mathcal{N} \rangle,$$

where $\mathcal{N}$ denotes the set of natural numbers.

Then the sequence $(1, 2, 3, 4)$ is a solution to this CSP since this sequence satisfies all constraints. Indeed, we have $1^3 + 2^3 + 3^3 + 4^3 = 100$, $1 < 4$ and $1 + 2 = 3$.

Finally, let us clarify one simple matter. When defining constraints and CSPs we refer to the sequences (respectively subsequences) of variables and *not* to the sets (respectively subsets) of variables. Namely, given a CSP each of its constraints is defined on a *subsequence* and not on a *subset* of its variables. In particular, the above constraint $x < y$ is defined on the subsequence x, y of the sequence x, y, z, u.

Also, the sequence z, y is not a subsequence x, y, z, u, so if we add to the above CSP the constraint $z = y + 2$ we cannot consider it as a constraint on z, y. But we can view it of course as a constraint on y, z and, if we wish,

we can rewrite it as $y + 2 = z$. The reliance on sequences and subsequences of variables instead of on sets and subsets will allow us to analyse in a simple way variable orderings when searching for solutions to a given CSP. So this presentation does not introduce any restrictions and simplifies some considerations.

2.2 Constraint satisfaction problems on integers

The remainder of this chapter is devoted to a presentation of various examples of constraint satisfaction problems. We begin with examples of CSPs with integer domains.

Example 2.1 *SEND + MORE = MONEY.*

This is a classic example of a so-called ***cryptarithmetic problem***. These are mathematical puzzles in which the digits are replaced by letters of the alphabet or other symbols. The problems dealing with valid sums are called ***alphametic*** problems. ***alphametic problem*** In the problem under consideration we are asked to replace each letter by a different digit so that the above sum, that is

$$\begin{array}{r} SEND \\ +\ MORE \\ \hline MONEY \end{array}$$

is correct. Here the variables are S, E, N, D, M, O, R, Y. Because S and M are the leading digits, the domain for each of them consists of the integer interval [1..9]. The domain of each of the remaining variables consists of the integer interval [0..9]. This problem can be formulated as the equality constraint

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

combined with 28 disequality constraints $x \neq y$ for x, y ranging over the set $\{S, E, N, D, M, O, R, Y\}$ with x preceding y in, say, the presented order.

A minor variation on the above representation of this problem as a CSP is obtained by assuming that the domains of all variables are the same, namely the interval [0..9], and by adding to the above constraints two disequality constraints:

$$S \neq 0,\ M \neq 0.$$

Yet another possibility consists of additionally introducing per column a 'carry' variable ranging over [0..1] and using instead of the above single equality constraint the following five, one for each column:

$$D + E = 10 \cdot C_1 + Y,$$

$$C_1 + N + R = 10 \cdot C_2 + E,$$

$$C_2 + E + O = 10 \cdot C_3 + N,$$

$$C_3 + S + M = 10 \cdot C_4 + O,$$

$$C_4 = M.$$

Here, $C_1, \ldots, C_4$ are the carry variables.

In turn, the disequality constraints can be replaced by a single constraint that stipulates that the variables are pairwise different. Given a sequence of variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$ we define

$$\texttt{all\_different}(x_1, \ldots, x_n) := \{(d_1, \ldots, d_n) \mid d_i \neq d_j \text{ for } i \neq j\}.$$

Then we can replace the above 28 disequality constraints by a single constraint $\texttt{all\_different}(S, E, N, D, M, O, R, Y)$.

Yet another way to deal with these disequality constraints is to follow a standard method used in the area of Integer Linear Programming, the objective of which is to find optimal integer solutions to linear constraints. For each pair x, y of different variables from $\{S, E, N, D, MO, R, Y\}$ we introduce a variable $z_{x,y}$ ranging over [0..1] and transform the disequality $x \neq y$ to the following two constraints:

- $x - y \leq 10 - 11z_{x,y}$,
- $y - x \leq 11z_{x,y} - 1$.

Note that $x \neq y$ is equivalent to the disjunction $x < y$ or $y < x$ which is equivalent to $x - y \leq -1$ or $y - x \leq -1$, which in turn is equivalent to the disjunction

$$(x - y \leq 10 - 11z_{x,y} \text{ and } z_{x,y} = 1) \text{ or } (y - x \leq 11z_{x,y} - 1 \text{ and } z_{x,y} = 0).$$

So to satisfy the constraint $x \neq y$ it is equivalent to satisfy one of the above two constraints on x, y and $z_{x,y}$. The disadvantage of this approach is that we need to introduce 28 new variables.

The above problem has a unique solution depicted by the following sum:

$$\begin{array}{r} 9567 \\ +\ 1085 \\ \hline 10652 \end{array}$$

As a consequence, each of the above representations of it as a CSP has a unique solution, as well. □

Example 2.2 *The n Queens Problem.*

This is probably the most known CSP. One is asked to place n queens on the $n \times n$ chess board, where $n \geq 3$, so that they do not attack each other. Figure 2.1 shows a solution to the problem for $n = 8$.

One possible representation of this problem as a CSP uses n variables, $x_1, \ldots, x_n$, each with the domain $[1..n]$. The idea is that x_i denotes the position of the queen placed in the ith column of the chess board. For example, the solution presented in Figure 2.1 corresponds to the sequence of values (6,4,7,1,8,2,5,3), since the first queen from the left is placed in the 6th row counting from the bottom, and similarly with the other queens.

The appropriate constraints can be formulated as the following disequalities for $i \in [1..n-1]$ and $j \in [i+1..n]$:

- $x_i \neq x_j$ (no two queens in the same row),
- $x_i - x_j \neq i - j$ (no two queens in each South-West – North-East diagonal),
- $x_i - x_j \neq j - i$ (no two queens in each North-West – South-East diagonal).

Using the `all_different` constraint introduced in the previous example we can replace the first set of $\frac{n \cdot (n-1)}{2}$ disequalities by a single constraint `all_different`$(x_1, \ldots, x_n)$. In Section 2.4 we shall discuss another natural representation of this problem. □

Example 2.3 *The Zebra Puzzle.*

As another example consider the following famous puzzle of Lewis Carroll. A small street has five differently coloured houses on it. Five men of different nationalities live in these five houses. Each man has a different profession, each man likes a different drink, and each has a different pet animal. We have the following information:

The Englishman lives in the red house.

The Spaniard has a dog.

The Japanese is a painter.

The Italian drinks tea.

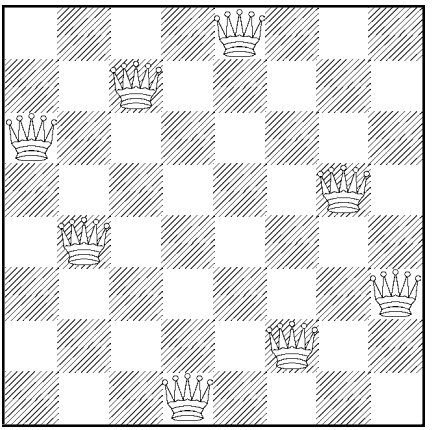

Fig. 2.1. One of 92 solutions to the 8 queens problem

The Norwegian lives in the first house on the left.
The owner of the green house drinks coffee.
The green house is on the right of the white house.
The sculptor breeds snails.
The diplomat lives in the yellow house.
They drink milk in the middle house.
The Norwegian lives next door to the blue house.
The violinist drinks fruit juice.
The fox is in the house next to the doctor's.
The horse is in the house next to the diplomat's.
The question is who has the zebra and who drinks water?

An interesting aspect of this puzzle is that in its data neither zebra nor water is mentioned and yet it has a unique solution. To formulate it as a CSP we first try to determine the variables and their domains. Note that this puzzle involves:

- five houses, which we number from left to right: 1, 2, 3, 4, 5,
- five colours, namely red, green, white, yellow, blue,
- five nationalities, namely English, Spanish, Japanese, Italian, Norwegian,
- five pets, namely dog, snails, fox, horse, and (implicitly) zebra,
- five professions, namely painter, sculptor, diplomat, violinist, doctor,
- five drinks, namely tea, coffee, milk, juice, and (implicitly) water.

To solve this puzzle it suffices to determine for each house its five characteristics:

- colour,
- nationality of the owner,
- pet of the owner,
- profession of the owner,
- favourite drink of the owner.

So we introduce 25 variables, five for each of the above five characteristics. For these variables we use the following mnemonic names:

- 'colour' variables: `red, green, white, yellow, blue`,
- 'nationality' variables: `english, spaniard, japanese, italian, norwegian`,
- 'pet' variables: `dog, snails, fox, horse, zebra`,
- 'profession' variables: `painter, sculptor, diplomat, violinist, doctor`,
- 'drink' variables: `tea, coffee, milk, juice, water`.

We assume that each of these variables ranges over $[1...5]$. If, for example, $\texttt{violinist} = 3$ then we interpret this as the statement that the violinist lives in house no. 3. After these preparations we can now formalise the given information as the following constraints:

- The Englishman lives in the red house: $\texttt{english} = \texttt{red}$,
- The Spaniard has a dog: $\texttt{spaniard} = \texttt{dog}$,
- The Japanese is a painter: $\texttt{japanese} = \texttt{painter}$,
- The Italian drinks tea: $\texttt{italian} = \texttt{tea}$,
- The Norwegian lives in the first house on the left: $\texttt{norwegian} = 1$,
- The owner of the green house drinks coffee: $\texttt{green} = \texttt{coffee}$,
- The green house is on the right of the white house: $\texttt{green} = \texttt{white} + 1$,
- The sculptor breeds snails: $\texttt{sculptor} = \texttt{snails}$,
- The diplomat lives in the yellow house: $\texttt{diplomat} = \texttt{yellow}$,
- They drink milk in the middle house: $\texttt{milk} = 3$,
- The Norwegian lives next door to the blue house: $|\texttt{norwegian} - \texttt{blue}| = 1$,
- The violinist drinks fruit juice: $\texttt{violinist} = \texttt{juice}$,
- The fox is in the house next to the doctor's: $|\texttt{fox} - \texttt{doctor}| = 1$,
- The horse is in the house next to the diplomat's: $|\texttt{horse} - \texttt{diplomat}| = 1$.

Additionally, we need to postulate that for each of the characteristics the corresponding variables are different. This means that we introduce fifty disequality constraints, ten for each characteristics. For example $\texttt{red} \neq$

white is one of such disequalities. Alternatively, instead of postulating these fifty disequality constraints we can employ the all_different constraint introduced in Example 2.1 and postulate instead

$$\texttt{all\_different}(\texttt{red}, \texttt{green}, \texttt{white}, \texttt{yellow}, \texttt{blue}),$$

$$\texttt{all\_different}(\texttt{english}, \texttt{spaniard}, \texttt{japanese}, \texttt{italian}, \texttt{norwegian}),$$

$$\texttt{all\_different}(\texttt{dog}, \texttt{snails}, \texttt{fox}, \texttt{horse}, \texttt{zebra}),$$

$$\texttt{all\_different}(\texttt{painter}, \texttt{sculptor}, \texttt{diplomat}, \texttt{violinist}, \texttt{doctor}),$$

$$\texttt{all\_different}(\texttt{tea}, \texttt{coffee}, \texttt{milk}, \texttt{juice}, \texttt{water}).$$

Now, it turns out that there is exactly one assignment of values to all 25 variables for which all constraints are satisfied. Because of this the puzzle has a unique solution Indeed, the puzzle is solved once we find in this unique assignment for which 'profession' variables

$$x, y \in \{\texttt{painter}, \texttt{sculptor}, \texttt{diplomat}, \texttt{violinist}, \texttt{doctor}\}$$

we have

$$x = \texttt{zebra} \text{ and } y = \texttt{water}.$$

The answer is $x = \texttt{japanese}$ and $y = \texttt{norwegian}$, which means that the Japanese has the zebra and the Norwegian drinks water. □

2.3 Constraint satisfaction problems on reals

Let us move now to the case of CSPs the variables of which range over reals.

Example 2.4 *Spreadsheets.*

Spreadsheet systems, such as Excel, are very popular for various office-like applications. These systems have in general a number of very advanced features but their essence relies on constraints.

To be more specific consider Table 2.1. It represents a spreadsheet, in which the values for the cells D4, D5, E7 and E8 are computed by means of the formulas present in these cells.

Equivalently, we can represent the spreadsheet of Table 2.1 by means of a CSP that consists of nine variables: B1, B4, B5, C4, C5, D4, D5, E7 and E8, each ranging over real numbers, and the following nine constraints:

B1 = 0.17,

B4 = 3.5,

	A	B	C	D	E
1	**Tax**	0.17			
2					
3	**Product**	**Price**	**Quantity**	**Total**	
4	tomatoes	3.5	1.5	= B4 ∗ C4	
5	potatoes	1.7	4.5	= B5 ∗ C5	
6					
7				**Grand Total**	= D4 + D5
8				**Final Amount**	= E7 ∗ (1 + B1)

Table 2.1. A spreadsheet

	A	B	C	D	E
1	**Tax**	0.17			
2					
3	**Product**	**Price**	**Quantity**	**Total**	
4	tomatoes	3.5	1.5	5.25	
5	potatoes	1.7	4.5	7.65	
6					
7				**Grand Total**	12.9
8				**Final Amount**	15.093

Table 2.2. Solution to the spreadsheet of Table 2.1

B5 = 1.7,
C4 = 1.5,
C5 = 4.5,
D4 = B4 ∗ C4,
D5 = B5 ∗ C5,
E7 = D4 + D5,
E8 = E7 ∗ (1 + B1).

A spreadsheet system solves these constraints and updates the solution each time a parameter is modified. The latter corresponds to modifying a numeric value v in a constraint of the form $x = v$, where x is a variable. For example, a modification of the C5 field corresponds to a modification of the right-hand side of the constraint C5 = 4.5.

In the case of the spreadsheet of Table 2.1 the solution is represented by the spreadsheet of Table 2.2. □

Example 2.5 *Finding zeros of polynomials of higher degree.*

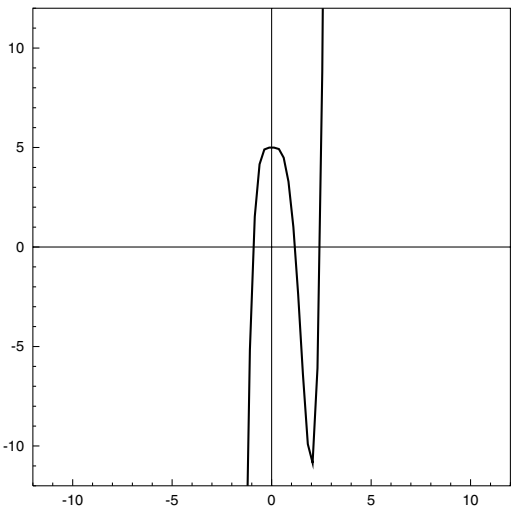

Fig. 2.2. The diagram of the polynomial $2 * x^5 - 5 * x^4 + 5$

Consider the polynomial $2*x^5-5*x^4+5$ represented in Figure 2.2. Suppose we wish to find its zeros. The difficulty lies in the fact that in general, by the celebrated Galois' theorem, the zeros of polynomials of degree higher than four cannot be expressed as radicals, that is by means of the four arithmetic operations and the root extraction. In fact, $2 * x^5 - 5 * x^4 + 5$ is one of such polynomials, found in the nineteenth century by the Norwegian mathematician Niels Henrik Abel. Another complication is that the real numbers cannot be faithfully represented in the computer.

The latter problem is dealt with by using the computer representable real numbers, i.e., the floating point numbers, and by producing solutions in the form of sequences of intervals with floating point bounds.

The techniques of constraint programming turn out to be useful to tackle such problems. In general, given a polynomial $f(x)$, we consider a CSP with a single constraint $f(x) = 0$ where the variable x ranges over reals. Further, we assume a fixed finite set of floating point numbers augmented with $-\infty$ and ∞, and denoted by F. By an ***interval CSP*** we mean here a CSP with the single constraint $f(x) = 0$ and a domain expression of the form $x \in [l, r]$, where $l, r \in F$, that is a CSP of the form $\langle f(x) = 0 \ ; \ x \in [l, r] \rangle$.

The original CSP is then transformed into a disjunction of the interval CSPs the intervals of which are of sizes smaller than some fixed in advance ϵ. This disjunction is equivalent to the original CSP, in the sense that the set of solutions to the original CSP equals the union of the sets of solutions to the disjunct interval CSPs.

In the case of the above polynomial, choosing the accuracy of 16 digits after the decimal point, we get the disjunction of three interval CSPs based on the following intervals:

$$x \in [-.9243580149260359080, \\ -.9243580149260359031],$$

$$x \in [1.171162068483181786, \\ 1.171162068483181791],$$

$$x \in [2.428072792707314923, \\ 2.428072792707314924].$$

An additional argument is needed to prove that each interval contains a zero. These considerations generalise to polynomials in an arbitrary number of variables and to the constrained optimization problems on reals according to which we are asked to optimize some real-valued function subject to polynomial constraints over the reals. □

2.4 Boolean constraint satisfaction problems

Once we identify **false** with 0 and **true** with 1, Boolean CSPs form a special case of numeric CSPs in which the variables range over the binary domain [0..1] and the constraints are expressed by means of Boolean expressions built using some basic set of connectives. From this perspective the first identified NP-complete problem, the satisfiability problem, according to which we ask whether a given Boolean expression is satisfiable, is equivalent to the question whether the corresponding Boolean CSP is consistent. In general, as we shall see in Chapter 6, the use of the constraint programming techniques does not bring any new insights into this area. However, these techniques allow us to model certain forms of reasoning in a natural way, also as programs.

Example 2.6 *The full adder circuit.*

This example deals with digital circuits built out of the AND, OR and XOR gates. These gates generate an output value given two input values.

The possible values are drawn from [0..1], so we deal here with bits or, alternatively, Boolean variables. The behaviour of these gates is defined by the following tables:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

Alternatively, we can view these gates as connectives: the AND gate corresponds to the conjunction, the OR gate to the disjunction and the XOR gate to the exclusive disjunction. Therefore the circuits built out of these gates can be naturally represented by equations between Boolean expressions involving conjunction, written as $\wedge$, disjunction, written as $\vee$, and exclusive disjunction, written as $\oplus$. In particular, the circuit depicted in Figure 2.3 can be represented by the following two equations:

$$(i_1 \oplus i_2) \oplus i_3 = o_1, \tag{2.1}$$

$$(i_1 \wedge i_2) \vee (i_3 \wedge (i_1 \oplus i_2)) = o_2. \tag{2.2}$$

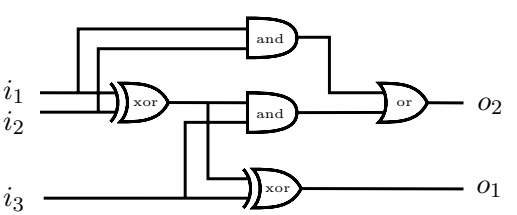

Fig. 2.3. Full adder circuit

This circuit is called ***full adder*** as it computes the binary sum $i_1 + i_2 + i_3$ in the binary word o_2o_1. For example 1+1+0 yields 10. To verify the correctness of this circuit it suffices to use the three tables above and to calculate using the equations (2.1) and (2.2) the outputs o_1, o_2 for all combinations of the inputs i_1, i_2, i_3.

The fact that we represent this circuit as a Boolean CSP, using relations between the Boolean variables i_1, i_2, i_3, o_1 and o_2 determined by the above equations, instead of as a function from (i_1, i_2, i_3) to (o_1, o_2), will allow us

in Section 6.3 to draw in a systematic way more complex conclusions such as that $i_3 = 0$ and $o_2 = 1$ implies that $i_1 = 1, i_2 = 1$ and $o_1 = 0$. □

Example 2.7 *The n Queens Problem, again.*

When discussing in Example 2.2 the n Queens Problem we chose a representation involving n variables, each associated with one row. A different natural representation involves n^2 Boolean variables $x_{i,j}$, where $i \in [1..n]$ and $j \in [1..n]$, each of them representing one field of the chess board. The appropriate constraints can then be written as Boolean constraints represented by Boolean expressions built using the conjunction and negation (written as $\neg$) connectives.

To this end we introduce the following abbreviation. Given k Boolean expressions $s_1, \ldots, s_k$ we denote by $one(s_1, \ldots, s_k)$ the Boolean expression that states that exactly one of the expressions $s_1, \ldots, s_k$ is true. So $one(s_1, \ldots, s_k)$ is a disjunction of k expressions, each of them being a k-ary conjunction of the form $\neg s_1 \wedge \ldots \neg s_{i-1} \wedge s_i \wedge \neg s_{i+1} \ldots \wedge \neg s_k$, where $i \in [1..k]$.

The following constraints then formalise the problem:

- $one(x_{i,1}, \ldots, x_{i,n})$ for $i \in [1..n]$ (exactly one queen per row),
- $one(x_{1,i}, \ldots, x_{n,i})$ for $i \in [1..n]$ (exactly one queen per column),
- $\neg(x_{i,j} \wedge x_{k,\ell})$ for $i, j, k, \ell \in [1..n]$ such that $i \neq k$ and $|i - k| = |j - \ell|$ (at most one queen per diagonal).

The condition $i \neq k$ in the last item ensures that the fields represented by $x_{i,j}$ and $x_{k,\ell}$ are different. □

2.5 Symbolic constraint satisfaction problems

By a symbolic constraint satisfaction problem we mean a CSP the variables of which range over non-numeric domains.

Example 2.8 *The Crossword Puzzle.*

Consider the crossword grid of Figure 2.4 and suppose that we are to fill it with the words from the following list:

- HOSES, LASER, SAILS, SHEET, STEER,
- HEEL, HIKE, KEEL, KNOT, LINE,
- AFT, ALE, EEL, LEE, TIE.

This problem can be formulated as a CSP as follows. First, associate with each position $i \in [1..8]$ in this grid a variable. Then associate with each variable the domain that consists of the set of words of that can be used to fill this position. For example, position 6 needs to be filled with a

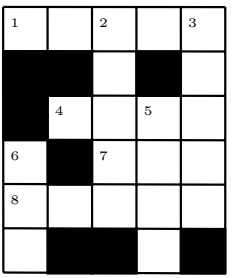

Fig. 2.4. A crossword grid

three letter word, so the domain of the variable associated with position 6 consists of the above set of five 3 letter words.

Finally, we define the constraints. They deal with the restrictions arising from the fact that the words that cross share a letter at appropriate positions. For example, the crossing of the positions 1 and 2 contributes the following constraint:

$$C_{1,2} := \{(\text{HOSES, SAILS}), (\text{HOSES, SHEET}), (\text{HOSES, STEER}), \\ (\text{LASER, SAILS}), (\text{LASER, SHEET}), (\text{LASER, STEER})\} .$$

This constraint formalises the fact that the third letter of position 1 needs to be the same as the first letter of position 2. In total there are twelve constraints. The unique solution to this CSP is depicted in Figure 2.5.

1 H	O	2 S	E	3 S
		A		T
	4 H	I	5 K	E
6 A		7 L	E	E
8 L	A	S	E	R
E			L	

Fig. 2.5. The solution to the crossword puzzle

□

The next two examples deal with ***qualitative reasoning***. In this form of reasoning one abstracts from the numeric quantities, such as the precise time of an event, or the location of an object in the space, and reasons instead on the level of their abstractions. Usually, these abstractions form a finite set

of alternatives, as opposed to the infinite set of possibilities available at the numeric level. This allows one to carry out conclusions on an abstract level that on the numeric level would be difficult to achieve. We now discuss two examples of qualitative reasoning.

Example 2.9 *Qualitative Temporal Reasoning.*

Consider the following problem.

The meeting ran non-stop the whole day. Each person stayed at the meeting for a continuous period of time. The meeting began while Mr Jones was present and finished while Ms White was present. Ms White arrived after the meeting has began. In turn, Director Smith was also present but he arrived after Jones had left. Mr Brown talked to Ms White in presence of Smith. Could possibly Jones and White have talked during this meeting?

To properly analyse such problems we are naturally led to an abstract analysis of activities that take time, such as being present during the meeting, talking to somebody, driving to work, taking a lunch break, filling in a form, receiving a phone call, etc. In what follows we call such activities ***temporal events***, or simply ***events***. If we only take into account the fact that such events take a continuous but limited period of time, then we can identify them with closed non-empty intervals of the real line.

In the case of the above problem a possible scenario that matches its description is provided in Figure 2.6, where each event (like the duration of the meeting or the period during which Jones was present) is represented by a closed non-empty real interval.

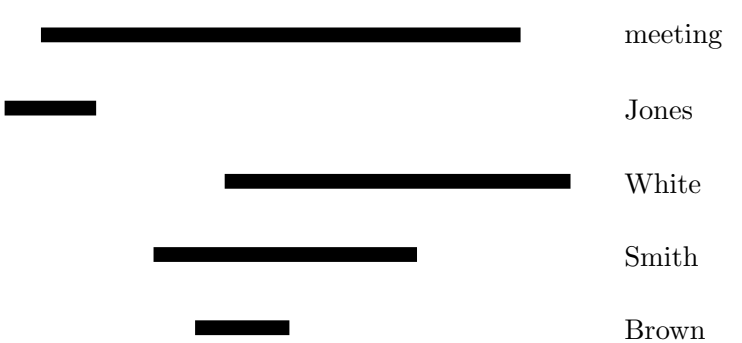

Fig. 2.6. A possible scenario for the 'meeting problem'

In general, when identifying events with closed non-empty real intervals we end up with thirteen possible ***temporal relations*** between a pair of events. They are presented in Figure 2.7. Intuitively, these relations arise when we consider two intervals, `A` and `B`, and keep moving the interval `A` from

left to right and record all its possible relative positions w.r.t. B, taking into account their possibly different relative sizes.

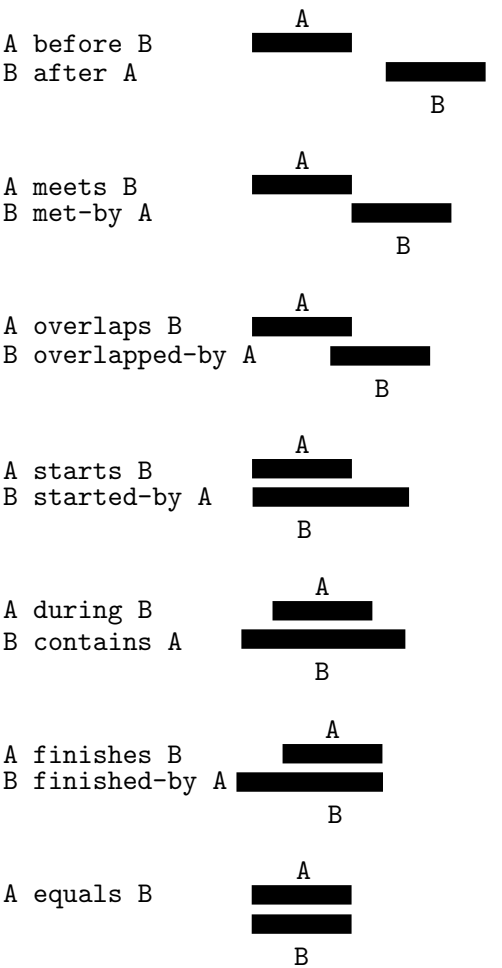

Fig. 2.7. Thirteen temporal relations

Let *TEMP* denote the set formed by the thirteen possibilities of Figure 2.7. In what follows we provide two representations of CSPs dealing with temporal reasoning in this setting. The first one is conceptually simpler but it involves infinite domains. The second one is at first appearance more involved but has the advantage that the domains are finite and directly correspond to the set *TEMP*.

First representation.

In this representation the variables represent events. Their domains reflect the view that events are identified with closed non-empty intervals of real line. So each domain consists of the set of such intervals, that is the set

$$D := \{(a, b) \mid a, b \in \mathcal{R},\ a < b\},$$

where $\mathcal{R}$ is the set of all reals and each pair (a, b) represents the closed interval $[a, b]$ of reals.

Next, each of the introduced thirteen temporal relations is represented as a binary constraint in the way that reflects the intended meaning. For example, the `overlaps` relation is represented as the following subset of $D \times D$:

$$[\![\texttt{overlaps}]\!] := \{((a_{begin}, a_{end}), (b_{begin}, b_{end})) \mid a_{begin} < b_{begin}, b_{begin} < a_{end}, a_{end} < b_{end}\}.$$

Finally, arbitrary constraints are set-theoretic unions of the elementary binary constraints that represent the thirteen temporal relations.

Let us illustrate now this representation by formalising as a CSP the problem we began with. First, we identify the relevant events and associate with each of them a variable. In total, we consider the following five events and variables:

event	variable
the duration of the meeting	M
the period Jones was present	J
the period Brown was present	B
the period Smith was present	S
the period White was present	W

Next, we define the appropriate constraints. The first three formalise information concerning the presence of each person during the meeting. For brevity we denote here the union of all elementary constraints excluding $[\![\texttt{before}]\!]$, $[\![\texttt{after}]\!]$, $[\![\texttt{meets}]\!]$ and $[\![\texttt{met-by}]\!]$ as $[\![\texttt{REAL-OVERLAP}]\!]$. Note that

$$[\![\texttt{REAL-OVERLAP}]\!] = \{((a_{begin}, a_{end}), (b_{begin}, b_{end})) \mid a_{begin} < b_{end}, b_{begin} < a_{end}\}.$$

These three constraints are:

$$([\![\texttt{overlaps}]\!] \cup [\![\texttt{contains}]\!] \cup [\![\texttt{finished-by}]\!])(J, M),$$

$$[\![\texttt{overlaps}]\!](M, W),$$

$$[\![\texttt{REAL-OVERLAP}]\!](M, S).$$

Note that the first constraint formalises the fact that

- J started strictly earlier than M started,
- M started strictly before J finished.

In turn, the second constraint formalises the fact that

- W started strictly before M finished,
- M finished strictly earlier than W finished,
- M started strictly before W started.

Finally, the constraint $[\![\texttt{REAL-OVERLAP}]\!](M, S)$ formalises the fact that M and S 'truly' overlap in time, that is, share some time interval of positive length. Note that we do not postulate any constraint on M and B, in particular not $[\![\texttt{REAL-OVERLAP}]\!](M, B)$, because from the problem formulation it is not clear whether Brown was actually present during the meeting.

Additionally, the following constraints formalise information concerning the relative presence of the persons in question:

$$[\![\texttt{before}]\!](J, S),$$

$$[\![\texttt{REAL-OVERLAP}]\!](B, S),$$

$$[\![\texttt{REAL-OVERLAP}]\!](B, W),$$

$$[\![\texttt{REAL-OVERLAP}]\!](S, W).$$

The question 'Could possibly Jones and White have talked during this meeting?' can now be formalised as a problem whether for some solution to this CSP $[\![\texttt{REAL} - \texttt{OVERLAP}]\!](J, W)$ holds. In other words, is it true that the above CSP augmented by the constraint $[\![\texttt{REAL} - \texttt{OVERLAP}]\!](J, W)$ is consistent.

Second representation.

In the first representation the domains, and thus the domain expressions, were uniquely fixed and we only had to determine the constraints all of which were binary. In the second representation we first determine the domain expressions which uniquely determine the constraints. In this representation a variable is associated with each ordered pair of events. Each domain of such a variable is a subset of the set *TEMP*. All constraints are ternary.

More specifically, consider three events, `A`,`B` and `C` and suppose that we know the temporal relations between the pairs `A` and `B`, and `B` and `C`. The

question is what is the temporal relation between A and C. For example if A overlaps B and B is before C, then A is before C. To answer this question we have to examine 169 possibilities. They are represented in a table called ***composition table***, given in Figures 2.8 and 2.9. The entry *R-OVERLAP* (an abbreviation for *REAL-OVERLAP*) which appears there three times, is a shorthand for the set of the temporal relations that express the fact that two events 'truly' overlap in time, so the set obtained from *TEMP* by excluding the relations before, after, meets and met-by:

$$\textit{REAL-OVERLAP} := \textit{TEMP} - \{\texttt{before, after, meets, met-by}\}.$$

Consider now all legally possible triples (t_1, t_2, t_3) of temporal relations such that t_1 is the temporal relation between A and B, t_2 the temporal relation between B and C and t_3 the temporal relation between A and C. The set of these triples forms a subset of $\textit{TEMP}^3$. Denote it by T_3. The just mentioned table allows us to compute T_3. For example, we already noticed that $(\texttt{overlaps}, \texttt{before}, \texttt{before}) \in T_3$, since A overlaps B and B is before C implies that A is before C. The set T_3 has 409 elements.

By a ***disjunctive temporal relation*** we mean now a disjunction of temporal relations. For example before $\vee$ meets is a disjunctive temporal relation. The disjunctive temporal relations allow us to model the situations in which the temporal dependencies between the events are only partially known.

Note that each disjunctive temporal relation between the events A and B uniquely determines the disjunctive temporal relation between the event B and A. For example if the former is before $\vee$ meets, then the latter is after $\vee$ met $-$ by. So without loss of information we can confine our attention to disjunctive temporal relations associated with each ordered pair of events.

We can now define the CSPs. Assume n events $e_1, \ldots, e_n$ with $n > 2$. We consider $\frac{(n-1)n}{2}$ domain expressions, each of the form $x_{i,j} \in D_{i,j}$ where $i, j \in [1..n]$ with $i < j$ and $D_{i,j} \subseteq \textit{TEMP}$. The intention is that the variable $x_{i,j}$ describes the disjunctive temporal relation associated with the events e_i and e_j.

Finally, we define the constraints. Each constraint links an ordered triple of events. So in total we have $\frac{(n-2)(n-1)n}{6}$ constraints. For $i, j, k \in [1..n]$ such that $i < j < k$ we define the constraint $C_{i,j,k}$ by setting

$$C_{i,j,k} := T_3 \cap (D_{i,j} \times D_{j,k} \times D_{i,k}).$$

So $C_{i,j,k}$ describes the possible entries in T_3 determined by the domains of the variables $x_{i,j}, x_{j,k}$ and $x_{i,k}$.

	before	after	meets	met-by	overlaps	overl.-by
before	before	*TEMP*	before	before meets overlaps starts during	before	before meets overlaps starts during
after	*TEMP*	after	during finishes after met-by overl.-by	after	during finishes after met-by overl.-by	after
meets	before	after met-by overl.-by started-by contains	before	finishes finished-by equals	before	overlaps starts during
met-by	before overlaps meets contains finished-by	after	starts started-by equals	after	during finishes overl.-by	after
overlaps	before	after met-by overl.-by started-by contains	before	overl.-by started-by contains	before meets overlaps	*R-OVERLAP*
overl.-by	before meets overlaps contains finished-by	after	overlaps contains finished-by	after	*R-OVERLAP*	after met-by overl.-by
starts	before	after	before	met-by	before meets overlaps	during finishes overl.-by
started-by	before meets overlaps contains finished-by	after	overlaps contains finished-by	met-by	overlaps contains finished-by	overl.-by
during	before	after	before	after	before meets overlaps starts during	during finishes after met-by overl.-by
contains	before meets overlaps contains finished-by	after met-by overl.-by contains started-by	overlaps contains finished-by	overl.-by started-by contains	overlaps contains finished-by	overl.-by started-by contains
finishes	before	after	meets	after	overlaps starts during	after met-by overl.-by
finished-by	before	after met-by overl.-by started-by contains	meets	overl.-by started-by contains	overlaps	overl.-by started-by contains
equals	before	after	meets	met-by	overlaps	overl.-by

Fig. 2.8. The composition table for the thirteen temporal relations, part 1

This completes the description of the CSPs. Because each constraint is determined by the corresponding triple of variable domains, each such CSP is uniquely determined by its domain expressions. Therefore when defining such CSPs it is sufficient to define the appropriate domain expressions.

Let us return now to the problem we started with. It can be formulated as a CSP as follows. We have five events, `M, J, B, S, W`. Here `M` stands for 'the duration of the meeting' , `J` stands for 'the period Jones was present',

	starts	started-by	during	contains	finishes	finished-by	equals
before	before	before	before meets overlaps starts during	before	before meets overlaps starts during	before	before
after	during finishes after met-by overl.-by	after	during finishes after met-by overl.-by	after	after	after	after
meets	meets	meets	overlaps starts during	before	overlaps starts during	before	meets
met-by	during finishes overl.-by	after	during finishes overl.-by	after	met-by	met-by	met-by
overlaps	overlaps	overlaps contains finished-by	overlaps starts during	before meets overlaps contains finished-by	overlaps starts during	before meets overlaps	overlaps
overl.-by	during finishes overl.-by	after met-by overl.-by	during finishes overl.-by	after meets overl.-by started-by contains	overl.-by	overl.-by started-by contains	overl.-by
starts	starts	starts started-by equals	during	before meets overlaps contains finished-by	during	before meets overlaps	starts
started-by	starts started-by equals	started-by	during finishes overl.-by	contains	overl.-by	contains	started-by
during	during	during finishes after met-by overl.-by	during	*TEMP*	during	before meets overlaps starts during	during
contains	overlaps contains finished-by	contains	*R-OVERLAP*	contains	overl.-by contains started-by	contains	contains
finishes	during	after met-by overl.-by	during	after met-by overl.-by started-by contains	finishes	finishes finished-by equals	finishes
finished-by	overlaps	contains	overlaps starts during	contains	finishes finished-by equals	finished-by	finished-by
equals	starts	started-by	during	contains	finishes	finished-by	equals

Fig. 2.9. The composition table for the thirteen temporal relations, part 2

and similarly with the events `S` (for Smith), `B` (for Brown) and `W` (for White). Next, we order the events in some arbitrary way, say

`J, M, B, S, W.`

We have in total ten variables, each associated with an ordered pair of events.

Analogously as in the first representation we use the relation that two events 'truly' overlap in time. It is now represented by the already introduced set *REAL-OVERLAP* of nine temporal relations.