Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Krzysztof R. Apt François Fages
Francesca Rossi Péter Szeredi
József Váncza (Eds.)

Recent Advances in Constraints

Joint ERCIM/CoLogNET International Workshop
on Constraint Solving and Constraint Logic Programming, CSCLP 2003
Budapest, Hungary, June 30 - July 2, 2003
Selected Papers

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editors

Krzysztof R. Apt
National University of Singapore, School of Computing
3 Science Drive 2, Republic of Singapore 117543
E-mail: apt@comp.nus.edu.sg

François Fages
INRIA Rocquencourt - Projet CONTRAINTES
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
E-mail: Francois.Fages@inria.fr

Francesca Rossi
University of Padua, Department of Pure and Applied Mathematics
Via G.B. Belzoni 7, 35131 Padua, Italy
E-mail: frossi@math.unipd.it

Péter Szeredi
Budapest University of Technology and Economics
Department of Computer Science and Information Theory
Magyar tudosok krt. 2, 1117 Budapest, Hungary
E-mail: szeredi@cs.bme.hu

József Váncza
Hungarian Academy of Sciences, Computer and Automation Research Institute
Kende u. 13-17, 1111 Budapest, Hungary
E-mail: vancza@sztaki.hu

Library of Congress Control Number: 2004103751

CR Subject Classification (1998): I.2.3-4, D.1, D.3.2-3, F.3.2, F.4.1, I.2.8, F.2.2

ISSN 0302-9743
ISBN 3-540-21834-3 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10997857 06/3142 5 4 3 2 1 0

Preface

Constraint programming is the fruit of several decades of research carried out in mathematical logic, automated deduction, operations research and artificial intelligence. The tools and programming languages arising from this research field have enjoyed real success in the industrial world as they contribute to solving hard combinatorial problems in diverse domains such as production planning, communication networks, robotics and bioinformatics.

This volume contains the extended and reviewed versions of a selection of papers presented at the Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2003), which was held from June 30 to July 2, 2003. The venue chosen for the seventh edition of this annual workshop was the Computer and Automation Research Institute of the Hungarian Academy of Sciences (MTA SZTAKI) in Budapest, Hungary. This institute is one of the 20 members of the Working Group on Constraints of the European Research Consortium for Informatics and Mathematics (ERCIM). For many participants this workshop provided the first opportunity to visit their ERCIM partner in Budapest.

CoLogNET is the European-funded network of excellence dedicated to supporting and enhancing cooperation and research on all areas of computational logic, and continues the work done previously by the Compulog Net. In particular, the aim of the logic and constraint logic programming area of CoLogNET is to foster and support all research activities related to logic programming and constraint logic programming.

The editors would like to take the opportunity and thank all the authors who submitted papers to this volume, as well as the reviewers for their helpful work. We hope this volume is useful for anyone interested in the recent advances and new trends in constraint programming, constraint solving, problem modelling and applications.

January 2004 K.R. Apt, F. Fages, F. Rossi, P. Szeredi and J. Váncza

Organization

CSCLP 2003 was organized by the ERCIM Working Group on Constraints, the CoLogNET area on Logic and Constraint Logic Programming, and the MTA SZTAKI in Budapest.

Organizing and Program Committee

Krzysztof Apt	University of Singapore, Singapore
François Fages	INRIA Rocquencourt, France
Francesca Rossi	University of Padua, Italy
Péter Szeredi	BUTE, Budapest, Hungary
József Váncza	MTA SZTAKI, Budapest, Hungary

Referees

K.R. Apt
R. Backofen
R. Barták
F. Benhamou
T. Benkő
S. Bistarelli
L. Bordeaux
S. Brand
M. Carlsson
M. Ceberio
N. Chabrier-Rivier
D. Diaz
F. Fages
B. Faltings
R. Gennari
W.J. van Hoeve
T. Kis
A. Lallouet
A. Márkus
E. Monfroy
B. O'Sullivan
G. Ringwelski
F. Rossi
H. Rudová
Zs. Ruttkay
H. Simonis
S. Soliman
P. Szeredi
J. Váncza
T. Walsh
A. Wolf

Sponsoring Institutions

ERCIM Working Group on Constraints
European Network of Excellence CoLogNET
Ministry of Education Hungary
MTA SZTAKI

Table of Contents

A Comparative Study of Arithmetic Constraints on Integer Intervals

Krzysztof R. Apt[1,2] and Peter Zoeteweij[1]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, the Netherlands
[2] University of Amsterdam, the Netherlands

Abstract. We propose here a number of approaches to implement constraint propagation for arithmetic constraints on integer intervals. To this end we introduce integer interval arithmetic. Each approach is explained using appropriate proof rules that reduce the variable domains. We compare these approaches using a set of benchmarks.

1 Preliminaries

1.1 Introduction

The subject of arithmetic constraints on reals has attracted a great deal of attention in the literature. For some reason arithmetic constraints on integer intervals have not been studied even though they are supported in a number of constraint programming systems. In fact, constraint propagation for them is present in ECL^{i}PS^e, SICStus Prolog, GNU Prolog, ILOG Solver and undoubtedly most of the systems that support constraint propagation for linear constraints on integer intervals. Yet, in contrast to the case of linear constraints — see notably [5] — we did not encounter in the literature any analysis of this form of constraint propagation.

In this paper we study these constraints in a systematic way. It turns out that in contrast to linear constraints on integer intervals there are a number of natural approaches to constraint propagation for these constraints.

To define them we introduce integer interval arithmetic that is modeled after the real interval arithmetic, see e.g., [6]. There are, however, essential differences since we deal with integers instead of reals. For example, multiplication of two integer intervals does not need to be an integer interval. In passing by we show that using integer interval arithmetic we can also define succinctly the well-known constraint propagation for linear constraints on integer intervals. In the second part of the paper we compare the proposed approaches by means of a set of benchmarks.

1.2 Constraint Satisfaction Problems

We review here the standard concepts of a constraint and of a constraint satisfaction problem. Consider a sequence of variables $X := x_1, \ldots, x_n$ where $n \geq 0$, with respective domains $D_1, \ldots, D_n$ associated with them. So each variable x_i ranges over the domain D_i. By a ***constraint*** C on X we mean a subset of $D_1 \times \ldots \times D_n$. Given an element $d := d_1, \ldots, d_n$ of $D_1 \times \ldots \times D_n$ and a subsequence $Y := x_{i_1}, \ldots, x_{i_\ell}$ of X we denote by $d[Y]$ the sequence $d_{i_1}, \ldots, d_{i_\ell}$. In particular, for a variable x_i from X, $d[x_i]$ denotes d_i.

A ***constraint satisfaction problem***, in short CSP, consists of a finite sequence of variables X with respective domains $\mathcal{D}$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write it as $\langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$, where $X := x_1, \ldots, x_n$ and $\mathcal{D} := D_1, \ldots, D_n$.

By a ***solution*** to $\langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ we mean an element $d \in D_1 \times \ldots \times D_n$ such that for each constraint $C \in \mathcal{C}$ on a sequence of variables X we have $d[X] \in C$. We call a CSP ***consistent*** if it has a solution and ***inconsistent*** if it does not. Two CSPs with the same sequence of variables are called ***equivalent*** if they have the same set of solutions. In what follows we consider CSPs the constraints of which are defined in a simple language and identify the syntactic description of a constraint with its meaning being the set of tuples that satisfy it.

We view ***constraint propagation*** as a process of transforming CSPs that maintains their equivalence. In what follows we define this process by means of proof rules that act of CSPs and preserve equivalence. An interested reader can consult [1] for a precise explanation of this approach to describing constraint propagation.

1.3 Arithmetic Constraints

To define the arithmetic constraints use the alphabet that comprises

– variables,
– two constants, 0 and 1,
– the unary minus function symbol '$-$',
– three binary function symbols, '$+$','$-$'and '$\cdot$', all written in the infix notation.

By an ***arithmetic expression*** we mean a term formed in this alphabet and by an ***arithmetic constraint*** a formula of the form

$$s \; op \; t,$$

where s and t are arithmetic expressions and $op \in \{<, \leq, =, \neq, \geq, >\}$. For example

$$x^5 \cdot y^2 \cdot z^4 + 3x \cdot y^3 \cdot z^5 \leq 10 + 4x^4 \cdot y^6 \cdot z^2 - y^2 \cdot x^5 \cdot z^4 \tag{1}$$

is an arithmetic constraint. Here x^5 is an abbreviation for $x \cdot x \cdot x \cdot x \cdot x$ and similarly with the other expressions. If '$\cdot$' is not used in an arithmetic constraint, we call it a ***linear constraint***.

By an ***extended arithmetic expression*** we mean a term formed in the above alphabet extended by the unary function symbols '$.^n$' and '$\sqrt[n]{.}$' for each

$n \geq 1$ and the binary function symbol '/' written in the infix notation. For example

$$\sqrt[3]{(y^2 \cdot z^4)/(x^2 \cdot u^5)} \tag{2}$$

is an extended arithmetic expression. Here, in contrast to the above x^5 is a term obtained by applying the function symbol '$\cdot^5$' to the variable x. The extended arithmetic expressions will be used only to define constraint propagation for the arithmetic constraints.

Fix now some arbitrary linear ordering $\prec$ on the variables of the language. By a ***monomial*** we mean an integer or a term of the form

$$a \cdot x_1^{n_1} \cdot \ldots \cdot x_k^{n_k}$$

where $k > 0$, $x_1, \ldots, x_k$ are different variables ordered w.r.t. $\prec$, and a is a non-zero integer and $n_1, \ldots, n_k$ are positive integers. We call then $x_1^{n_1} \cdot \ldots \cdot x_k^{n_k}$ the ***power product*** of this monomial.

Next, by a ***polynomial*** we mean a term of the form

$$\Sigma_{i=1}^{n} m_i,$$

where $n > 0$, at most one monomial m_i is an integer, and the power products of the monomials $m_1, \ldots, m_n$ are pairwise different. Finally, by a ***polynomial constraint*** we mean an arithmetic constraint of the form $s \ op \ b$, where s is a polynomial with no monomial being an integer, $op \in \{<, \leq, =, \neq, \geq, >\}$, and b is an integer. It is clear that by means of appropriate transformation rules we can transform each arithmetic constraint to a polynomial constraint. For example, assuming the ordering $x \prec y \prec z$ on the variables, the arithmetic constraint (1) can be transformed to the polynomial constraint

$$2x^5 \cdot y^2 \cdot z^4 - 4x^4 \cdot y^6 \cdot z^2 + 3x \cdot y^3 \cdot z^5 \leq 10$$

So, without loss of generality, from now on we shall limit our attention to the polynomial constraints.

Next, let us discuss the domains over which we interpret the arithmetic constraints. By an ***integer interval***, or an ***interval*** in short, we mean an expression of the form

$$[a..b]$$

where a and b are integers; $[a..b]$ denotes the set of all integers between a and b, including a and b. If $a > b$, we call $[a..b]$ the ***empty interval*** and denote it by $\emptyset$. Finally, by a ***range*** we mean an expression of the form

$$x \in I$$

where x is a variable and I is an interval.

2 Integer Set Arithmetic

To reason about the arithmetic constraints we employ a generalization of the arithmetic operations to the sets of integers.

2.1 Definitions

For X, Y sets of integers we define the following operations:

– addition:
$$X + Y := \{x + y \mid x \in X, y \in Y\},$$
– subtraction:
$$X - Y := \{x - y \mid x \in X, y \in Y\},$$
– multiplication:
$$X \cdot Y := \{x \cdot y \mid x \in X, y \in Y\},$$
– division:
$$X/Y := \{u \in \mathcal{Z} \mid \exists x \in X \exists y \in Y \; u \cdot y = x\},$$
– exponentiation:
$$X^n := \{x^n \mid x \in X\},$$
for each natural number $n > 0$,
– root extraction:
$$\sqrt[n]{X} := \{x \in \mathcal{Z} \mid x^n \in X\},$$
for each natural number $n > 0$.

All the operations except division are defined in the expected way. We shall return to it at the end of Section 6. At the moment it suffices to note the division operation is defined for all sets of integers, including $Y = \emptyset$ and $Y = \{0\}$. This division operation corresponds to the following division operation on the sets of reals introduced in [8]:

$$\frac{X}{Y} := \{u \in \mathcal{R} \mid \exists x \in X \exists y \in Y \; u \cdot y = x\}.$$

For a (n integer or real) number a and $op \in \{+, -, \cdot, /\}$ we identify $a \, op \, X$ with $\{a\} \, op \, X$ and $X \, op \, a$ with $X \, op \, \{a\}$.

To present the rules we are interested in we shall also use the addition and division operations on the sets of real numbers. Addition is defined in the same way as for the sets of integers, and division is defined above. In [6] it is explained how to implement these operations.

Further, given a set A of integers or reals, we define

$$^{\leq}A := \{x \in \mathcal{Z} \mid \exists a \in A \; x \leq a\},$$

$$^{\geq}A := \{x \in \mathcal{Z} \mid \exists a \in A \; x \geq a\}.$$

When limiting our attention to intervals of integers the following simple observation is of importance.

Note 1. For X, Y integer intervals and a an integer the following holds:

- $X \cap Y$, $X + Y, X - Y$ are integer intervals.
- $X/\{a\}$ is an integer interval.
- $X \cdot Y$ does not have to be an integer interval, even if $X = \{a\}$ or $Y = \{a\}$.
- X/Y does not have to be an integer interval.
- For each $n > 1$ X^n does not have to be an integer interval.
- For odd $n > 1$ $\sqrt[n]{X}$ is an integer interval.
- For even $n > 1$ $\sqrt[n]{X}$ is an integer interval or a disjoint union of two integer intervals. □

For example we have

$$[2..4] + [3..8] = [5..12],$$

$$\begin{aligned}
[3..7] - [1..8] &= [-5..6],\\
[3..3] \cdot [1..2] &= \{3, 6\},\\
[3..5]/[-1..2] &= \{-5, -4, -3, 2, 3, 4, 5\},\\
[-3..5]/[-1..2] &= \mathcal{Z},\\
[1..2]^2 &= \{1, 4\},\\
\sqrt[3]{[-30..100]} &= [-3..4],\\
\sqrt[2]{[-100..9]} &= [-3..3],\\
\sqrt[2]{[1..9]} &= [-3..-1] \cup [1..3].
\end{aligned}$$

To deal with the problem that non-interval domains can be produced by some of the operations we introduce the following operation on the subsets of the set of the integers $\mathcal{Z}$:

$$int(X) := \begin{cases} \text{smallest integer interval containing } X & \text{if } X \text{ is finite,} \\ \mathcal{Z} & \text{otherwise.} \end{cases}$$

For example $int([3..5]/[-1..2]) = [-5..5]$ and $int([-3..5]/[-1..2]) = \mathcal{Z}$.

2.2 Implementation

To define constraint propagation for the arithmetic constraints on integer intervals we shall use the integer set arithmetic, mainly limited to the integer intervals. This brings us to the discussion of how to implement the introduced operations on the integer intervals. Since we are only interested in maintaining the property that the sets remain integer intervals or the set of integers $\mathcal{Z}$ we shall clarify how to implement the intersection, addition, subtraction and root extraction operations of the integer intervals and the $int(.)$ closure of the multiplication, division and exponentiation operations on the integer intervals. The case when one of the intervals is empty is easy to deal with. So we assume that we deal with non-empty intervals $[a..b]$ and $[c..d]$, that is $a \leq b$ and $c \leq d$.

Intersection, addition and subtraction. It is easy to see that

$$\begin{aligned}[a..b] \cap [c..d] &= [max(a,c)..min(b,d)],\\ [a..b] + [c..d] &= [a + c \,..\, b + d],\\ [a..b] - [c..d] &= [a - d \,..\, b - c].\end{aligned}$$

So the interval intersection, addition, and subtraction are straightforward to implement.

Root extraction. The outcome of the root extraction operator applied to an integer interval will be an integer interval or a disjoint union of two integer intervals. We shall explain in Section 4 why it is advantageous not to apply *int*(.) to the outcome. This operator can be implemented by means of the following case analysis.

Case 1. Suppose n is odd. Then

$$\sqrt[n]{[a..b]} = [\lceil \sqrt[n]{a} \rceil \,..\, \left\lfloor \sqrt[n]{b} \right\rfloor].$$

Case 2. Suppose n is even and $b < 0$. Then

$$\sqrt[n]{[a..b]} = \emptyset.$$

Case 3. Suppose n is even and $b \geq 0$. Then

$$\sqrt[n]{[a..b]} = [-\left\lfloor |\sqrt[n]{b}| \right\rfloor \,..\, -\left\lceil |\sqrt[n]{a^+}| \right\rceil] \cup [\left\lceil |\sqrt[n]{a^+}| \right\rceil \,..\, \left\lfloor |\sqrt[n]{b}| \right\rfloor]$$

where $a^+ := max(0, a)$.

Multiplication. For the remaining operations we only need to explain how to implement the *int*(.) closure of the outcome. First note that

$$int([a..b] \cdot [c..d]) = [min(A)..max(A)],$$

where $A = \{a \cdot c, a \cdot d, b \cdot c, b \cdot d\}$.

Using an appropriate case analysis we can actually compute the bounds of $int([a..b] \cdot [c..d])$ directly in terms of the bounds of the constituent intervals.

Division. In contrast, the *int*(.) closure of the interval division is not so straightforward to compute. The reason is that, as we shall see in a moment, we cannot express the result in terms of some simple operations on the interval bounds.

Consider non-empty integer intervals $[a..b]$ and $[c..d]$. In analyzing the outcome of $int([a..b]/[c..d])$ we distinguish the following cases.

Case 1. Suppose $0 \in [a..b]$ and $0 \in [c..d]$.

Then by definition $int([a..b]/[c..d]) = \mathcal{Z}$. For example,

$$int([-1..100]/[-2..8]) = \mathcal{Z}.$$

Case 2. Suppose $0 \notin [a..b]$ and $c = d = 0$.
Then by definition $int([a..b]/[c..d]) = \emptyset$. For example,

$$int([10..100]/[0..0]) = \emptyset.$$

Case 3. Suppose $0 \notin [a..b]$ and $c < 0$ and $0 < d$.
It is easy to see that then

$$int([a..b]/[c..d]) = [-e..e],$$

where $e = max(|a|, |b|)$. For example,

$$int([-100..-10]/[-2..5]) = [-100..100].$$

Case 4. Suppose $0 \notin [a..b]$ and either $c = 0$ and $d \neq 0$ or $c \neq 0$ and $d = 0$.
Then $int([a..b]/[c..d]) = int([a..b]/([c..d] - \{0\}))$. For example

$$int([1..100]/[-7..0]) = int([1..100]/[-7..-1]).$$

This allows us to reduce this case to Case 5 below.
Case 5. Suppose $0 \notin [c..d]$.
This is the only case when we need to compute $int([a..b]/[c..d])$ indirectly. First, observe that we have

$$int([a..b]/[c..d]) \subseteq [\lceil min(A) \rceil \,..\, \lfloor max(A) \rfloor],$$

where $A = \{a/c, a/d, b/c, b/d\}$.

However, the equality does not need to hold here. Indeed, note for example that $int([155..161]/[9..11]) = [16..16]$, whereas for $A = \{155/9, 155/11, 161/9, 161/11\}$ we have $\lceil min(A) \rceil = 15$ and $\lfloor max(A) \rfloor = 17$. The problem is that the value 16 is obtained by dividing 160 by 10 and none of these two values is an interval bound.

This complication can be solved by preprocessing the interval $[c..d]$ so that its bounds are actual divisors of an element of $[a..b]$. First, we look for the least $c' \in [c..d]$ such that $\exists x \in [a..b]\ \exists u \in \mathcal{Z}\ u \cdot c' = x$. Using a case analysis, the latter property can be established without search. Suppose for example that $a > 0$ and $c > 0$. In this case, if $c' \cdot \lfloor \frac{b}{c'} \rfloor \geq a$, then c' has the required property. Similarly, we look for the largest $d' \in [c..d]$ for which an analogous condition holds. Now $int([a..b]/[c..d]) = [\lceil min(A) \rceil .. \lfloor max(A) \rfloor]$, where $A = \{a/c', a/d', b/c', b/d'\}$.

Exponentiation. The $int(.)$ closure of the interval exponentiation is straightforward to implement by distinguishing the following cases.

Case 1. Suppose n is odd. Then

$$int([a..b]^n) = [a^n .. \, b^n].$$

Case 2. Suppose n is even and $0 \leq a$. Then

$$int([a..b]^n) = [a^n .. \, b^n].$$

Case 3. Suppose n is even and $b \leq 0$. Then

$$int([a..b]^n) = [b^n .. \, a^n].$$

Case 4. Suppose n is even and $a < 0$ and $0 < b$. Then

$$int([a..b]^n) = [0..max(a^n, b^n)].$$

2.3 Correctness Lemma

Given now an extended arithmetic expression s each variable of which ranges over an integer interval, we define $int(s)$ as the integer interval or the set $\mathcal{Z}$ obtained by systematically replacing each function symbol by the application of the $int(.)$ operation to the corresponding integer set operation. For example, for the extended arithmetic expression $s := \sqrt[3]{(y^2 \cdot z^4)/(x^2 \cdot u^5)}$ of (2) we have

$$int(s) = int(\sqrt[3]{int(int(Y^2) \cdot int(Z^4))/int(int(X^2) \cdot int(U^5))}),$$

where x ranges over X, etc.

The discussion in the previous subsection shows how to compute $int(s)$ given an extended arithmetic expression s and the integer interval domains of its variables.

The following lemma is crucial for our considerations. It is a counterpart of the so-called 'Fundamental Theorem of Interval Arithmetic' established in [7]. Because we deal here with the integer domains an additional assumption is needed to establish the desired conclusion.

Lemma 1 (Correctness). *Let s be an extended arithmetic expression with the variables $x_1, \ldots, x_n$. Assume that each variable x_i of s ranges over an integer interval X_i. Choose $a_i \in X_i$ for $i \in [1..n]$ and denote by $s(a_1, \ldots, a_n)$ the result of replacing in s each occurrence of a variable x_i by a_i.*

Suppose that each subexpression of $s(a_1, \ldots, a_n)$ evaluates to an integer. Then the result of evaluating $s(a_1, \ldots, a_n)$ is an element of $int(s)$.

Proof. The proof follows by a straightforward induction on the structure of s. □

3 An Intermezzo: Constraint Propagation for Linear Constraints

Even though we focus here on arithmetic constraints on integer intervals, it is helpful to realize that the integer interval arithmetic is also useful to define in a succinct way the well-known rules for constraint propagation for linear constraints. To this end consider first a constraint $\Sigma_{i=1}^{n} a_i \cdot x_i = b$, where $n \geq 0$, $a_1, \ldots, a_n$ are non-zero integers, $x_1, \ldots, x_n$ are different variables, and b is an integer. To reason about it we can use the following rule parametrized by $j \in [1..n]$:

LINEAR EQUALITY

$$\frac{\langle \Sigma_{i=1}^{n} a_i \cdot x_i = b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \Sigma_{i=1}^{n} a_i \cdot x_i = b \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \rangle}$$

where

– for $i \neq j$
$$D'_i := D_i,$$

–
$$D'_j := D_j \cap int\Big((b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot x_i)/a_j\Big).$$

Note that by virtue of Note 1

$$D'_j = D_j \cap (b - \Sigma_{i \in [1..n]-\{j\}} int(a_i \cdot D_i))/a_j.$$

To see that this rule preserves equivalence suppose that for some $d_1 \in D_1, \ldots, d_n \in D_n$ we have $\Sigma_{i=1}^{n} a_i \cdot d_i = b$. Then for $j \in [1..n]$ we have

$$d_j = (b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot d_i)/a_j$$

which by the Correctness Lemma 1 implies that

$$d_j \in int\Big((b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot x_i)/a_j\Big),$$

i.e., $d_j \in D'_j$.

Next, consider a constraint $\Sigma_{i=1}^{n} a_i \cdot x_i \leq b$, where $a_1, \ldots, a_n, x_1, \ldots, x_n$ and b are as above. To reason about it we can use the following rule parametrized by $j \in [1..n]$:

LINEAR INEQUALITY

$$\frac{\langle \Sigma_{i=1}^{n} a_i \cdot x_i \leq b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \Sigma_{i=1}^{n} a_i \cdot x_i \leq b \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \rangle}$$

where

– for $i \neq j$
$$D'_i := D_i,$$

–
$$D'_j := D_j \cap (^{\leq} int(b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot x_i)/a_j)$$

To see that this rule preserves equivalence suppose that for some $d_1 \in D_1, \ldots, d_n \in D_n$ we have $\Sigma_{i=1}^{n} a_i \cdot d_i \leq b$. Then $a_j \cdot d_j \leq b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot d_i$. By the Correctness Lemma 1

$$b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot d_i \in int(b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot x_i),$$

so by definition

$$a_j \cdot d_j \in^{\leq} int(b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot x_i)$$

and consequently

$$d_j \in^{\leq} int(b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot x_i)/a_j$$

This implies that $d_j \in D'_j$.

4 Constraint Propagation: First Approach

We now move on to a discussion of constraint propagation for the arithmetic constraints on integer intervals. To illustrate the first approach consider the following example. Consider the constraint

$$x^3y - x \le 40$$

and the ranges $x \in [1..100]$ and $y \in [1..100]$. We can rewrite it as

$$x \le \left\lfloor \sqrt[3]{\frac{40+x}{y}} \right\rfloor \tag{3}$$

since x assumes integer values. The maximum value the expression on the right-hand side can take is $\lfloor \sqrt[3]{140} \rfloor$, so we conclude $x \le 5$. By reusing (3), now with the information that $x \in [1..5]$, we conclude that the maximum value the expression on the right-hand side of (3) can take is actually $\lfloor \sqrt[3]{45} \rfloor$, from which it follows that $x \le 3$.

In the case of y we can isolate it by rewriting the original constraint as $y \le \frac{40}{x^3} + \frac{1}{x^2}$ from which it follows that $y \le 41$, since by assumption $x \ge 1$. So we could reduce the domain of x to $[1..3]$ and the domain of y to $[1..41]$. This interval reduction is optimal, since $x = 1, y = 41$ and $x = 3, y = 1$ are both solutions to the original constraint $x^3y - x \le 40$.

More formally, we consider a polynomial constraint $\Sigma_{i=1}^{m} m_i = b$ where $m > 0$, no monomial m_i is an integer, the power products of the monomials are pairwise different and b is an integer. Suppose that $x_1, \ldots, x_n$ are its variables ordered w.r.t. $\prec$.

Select a non-integer monomial m_ℓ and assume it is of the form

$$a \cdot y_1^{n_1} \cdot \ldots \cdot y_k^{n_k},$$

where $k > 0$, $y_1, \ldots, y_k$ are different variables ordered w.r.t. $\prec$, a is a non-zero integer and $n_1, \ldots, n_k$ are positive integers. So each y_i variable equals to some variable in $\{x_1, \ldots, x_n\}$. Suppose that y_p equals to x_j. We introduce the following proof rule:

POLYNOMIAL EQUALITY

$$\frac{\langle \Sigma_{i=1}^{n} m_i = b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \Sigma_{i=1}^{n} m_i = b \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \rangle}$$

where

– for $i \ne j$

$$D'_i := D_i,$$

–

$$D'_j := int\left(D_j \cap \sqrt[n_p]{int\left((b - \Sigma_{i \in [1..m]-\{\ell\}} m_i)/s \right)} \right)$$

and

$$s := a \cdot y_1^{n_1} \cdot \ldots \cdot y_{p-1}^{n_{p-1}} \cdot y_{p+1}^{n_{p+1}} \cdot \ldots \cdot y_k^{n_k}.$$

To see that this rule preserves equivalence choose some $d_1 \in D_1, \ldots, d_n \in D_n$. To simplify the notation, given an extended arithmetic expression t denote by t' the result of evaluating t after each occurrence of a variable x_i is replaced by d_i.

Suppose that $\Sigma_{i=1}^{m} m_i' = b$. Then

$$d_j^{n_p} \cdot s' = b - \Sigma_{i \in [1..m]-\{\ell\}} m_i',$$

so by the Correctness Lemma 1 applied to $b - \Sigma_{i \in [1..m]-\{\ell\}} m_i'$ and to s

$$d_j^{n_p} \in int(b - \Sigma_{i \in [1..m]-\{\ell\}} m_i)/int(s).$$

Hence

$$d_j \in \sqrt[n_p]{int(b - \Sigma_{i \in [1..m]-\{\ell\}} m_i)/int(s)}$$

and consequently

$$d_j \in int\left(D_j \cap \sqrt[n_p]{int\left((b - \Sigma_{i \in [1..m]-\{\ell\}} m_i)/s\right)}\right)$$

i.e., $d_j \in D_j'$.

Note that we do not apply $int(.)$ to the outcome of the root extraction operation. For even n_p this means that the second operand of the intersection can be a union of two intervals, instead of a single interval. To see why this is desirable, consider the constraint $x^2 - y = 0$ in the presence of ranges $x \in [0..10]$, $y \in [25..100]$. Using the $int(.)$ closure of the root extraction we would not be able to update the lower bound of x to 5.

Next, consider a polynomial constraint $\Sigma_{i=1}^{m} m_i \leq b$. Below we adopt the assumptions and notation used when defining the *POLYNOMIAL EQUALITY* rule. To formulate the appropriate rule we stipulate that for extended arithmetic expressions s and t

$$int((^{\leq}s)/t) \ := \ ^{\geq}Q \cap \ ^{\leq}Q,$$

with $Q = (^{\leq}int(s))/int(t)$.

To reason about this constraint we use the following rule:

POLYNOMIAL INEQUALITY

$$\frac{\langle \Sigma_{i=1}^{n} m_i \leq b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \Sigma_{i=1}^{n} m_i \leq b \ ; \ x_1 \in D_1', \ldots, x_n \in D_n' \rangle}$$

where

– for $i \neq j$

$$D_i' := D_i,$$

–

$$D_j' := int\left(D_j \cap \sqrt[n_p]{int\left(^{\leq}(b - \Sigma_{i \in [1..m]-\{\ell\}} m_i)/s\right)}\right)$$

To prove that this rule preserves equivalence choose some $d_1 \in D_1, \ldots, d_n \in D_n$. As above given an extended arithmetic expression t we denote by t' the result of evaluating t when each occurrence of a variable x_i in t is replaced by d_i.

Suppose that $\Sigma_{i=1}^m m'_i \leq b$. Then

$$d_j^{n_p} \cdot s' \leq b - \Sigma_{i \in [1..m]-\{\ell\}} m'_i.$$

By the Correctness Lemma 1

$$b - \Sigma_{i \in [1..m]-\{\ell\}} m'_i \in int(b - \Sigma_{i \in [1..m]-\{\ell\}} m_i),$$

so by definition

$$d_j^{n_p} \cdot s' \in^{\leq} int(b - \Sigma_{i \in [1..m]-\{\ell\}} m_i).$$

Hence by the definition of the division operation on the sets of integers

$$d_j^{n_p} \in^{\leq} int(b - \Sigma_{i \in [1..m]-\{\ell\}} m_i)/int(s)$$

Consequently

$$d_j \in \sqrt[n_p]{{}^{\leq}int(b - \Sigma_{i \in [1..m]-\{\ell\}} m_i)/int(s)}$$

This implies that $d_j \in D'_j$.

Note that the set ${}^{\leq}int(b - \Sigma_{i \in [1..m]-\{\ell\}} m_i)$ is not an interval. So to properly implement this rule we need to extend the implementation of the division operation discussed in Subsection 2.2 to the case when the numerator is an extended interval. Our implementation takes care of this.

In an optimized version of this approach we simplify the fractions of two polynomials by splitting the division over addition and subtraction and by dividing out common powers of variables and greatest common divisors of the constant factors. Subsequently, fractions whose denominators have identical power products are added. We used this optimization in the initial example by simplifying $\frac{40+x}{x^3}$ to $\frac{40}{x^3} + \frac{1}{x^2}$. The reader may check that without this simplification step we can only deduce that $y \leq 43$.

To provide details of this optimization, given two monomials s and t, we denote by

$$[\frac{s}{t}]$$

the result of performing this simplification operation on s and t. For example, $[\frac{2 \cdot x^3 \cdot y}{4 \cdot x^2}]$ equals $\frac{x \cdot y}{2}$, whereas $[\frac{4 \cdot x^3 \cdot y}{2 \cdot y^2}]$ equals $\frac{2 \cdot x^3}{y}$.

In this approach we assume that the domains of the variables $y_1, \ldots, y_{p-1}$, $y_{p+1}, \ldots, y_n$ of m_ℓ do not contain 0. (One can easily show that this restriction is necessary here). For a monomial s involving variables ranging over the integer intervals that do not contain 0, the set $int(s)$ either contains only positive numbers or only negative numbers. In the first case we write $sign(s) = +$ and in the second case we write $sign(s) = -$.

The new domain of the variable x_j in the *POLYNOMIAL INEQUALITY* rule is defined using two sequences $m'_0...m'_n$ and $s'_0...s'_n$ of extended arithmetic expressions such that

$$\frac{m'_0}{s'_0} = [\frac{b}{s}] \text{ and } \frac{m'_i}{s'_i} = -[\frac{m_i}{s}] \text{ for } i \in [1..m].$$

Let $S := \{s'_i \mid i \in [0..m] - \{\ell\}\}$ and for an extended arithmetic expression $t \in S$ let $I_t := \{i \in [0..m] - \{\ell\} \mid s'_i = t\}$. We denote then by p_t the polynomial $\sum_{i \in I_t} m'_i$. The new domains are then defined by

$$D'_j := int\left(D_j \cap \sqrt[n_p]{{}^{\leq}int\left(\Sigma_{t \in S}\ \frac{p_t}{t}\right)}\ \right)$$

if $sign(s) = +$, and by

$$D'_j := int\left(D_j \cap \sqrt[n_p]{{}^{\geq}int\left(\Sigma_{t \in S}\ \frac{p_t}{t}\right)}\ \right)$$

if $sign(s) = -$. Here the $int(s)$ notation used in the Correctness Lemma 1 is extended to expressions involving the division operator on real intervals in the obvious way. We define the $int(.)$ operator applied to a bounded set of real numbers, as produced by the division and addition operators in the above two expressions for D'_j, to denote the smallest interval of real numbers containing that set.

5 Constraint Propagation: Second Approach

In this approach we limit our attention to a special type of polynomial constraints, namely the ones of the form $s\ op\ b$, where s is a polynomial in which each variable occurs *at most once* and where b is an integer. We call such a constraint a *simple polynomial constraint*. By introducing auxiliary variables that are equated with appropriate monomials we can rewrite each polynomial constraint into a sequence of simple polynomial constraints. This allows us also to compute the integer interval domains of the auxiliary variable from the integer interval domains of the original variables. We apply then to the simple polynomial constraints the rules introduced in the previous section.

To see that the restriction to simple polynomial constraints can make a difference consider the constraint

$$100x \cdot y - 10y \cdot z = 212$$

in presence of the ranges $x, y, z \in [1..9]$. We rewrite it into the sequence

$$u = x \cdot y,\ v = y \cdot z,\ 100u - 10v = 212$$

where u, v are auxiliary variables, each with the domain $[1..81]$.

It is easy to check that the *POLYNOMIAL EQUALITY* rule introduced in the previous section does not yield any domain reduction when applied to the original constraint $100x \cdot y - 10y \cdot z = 212$. In presence of the discussed optimization the domain of x gets reduced to $[1..3]$.

However, if we repeatedly apply the *POLYNOMIAL EQUALITY* rule to the simple polynomial constraint $100u - 10v = 212$, we eventually reduce the domain of u to the empty set (since this constraint has no integer solution in the ranges $u, v \in [1..81]$) and consequently can conclude that the original constraint $100x \cdot y - 10y \cdot z = 212$ has no solution in the ranges $x, y, z \in [1..9]$, without performing any search.

6 Constraint Propagation: Third Approach

In this approach we focus on a small set of 'atomic' arithmetic constraints. We call an arithmetic constraint ***atomic*** if it is in one of the following two forms:

– a linear constraint,
– $x \cdot y = z$.

It is easy to see that using appropriate transformation rules involving auxiliary variables we can transform each arithmetic constraint to a sequence of atomic arithmetic constraints. In this transformation, as in the second approach, the auxiliary variables are equated with monomials so we can easily compute their domains.

The transformation to atomic constraints can strengthen the reduction. Consider for example the constraint

$$u \cdot x \cdot y + 1 = v \cdot x \cdot y$$

and ranges $u \in [1..2]$, $v \in [3..4]$, and $x, y \in [1..4]$. The first approach without optimization and the second approach cannot find a solution without search. If, as a first step in transforming this constraint into a linear constraint, we introduce an auxiliary variable w to replace $x \cdot y$, we are effectively solving the constraint

$$u \cdot w + 1 = v \cdot w$$

with the additional range $w \in [1..16]$, resulting in only one duplicate occurrence of a variable instead of two. With variable w introduced (or using the optimized version of the first approach) constraint propagation alone finds the solution $u = 2$, $v = 3$, $x = 1$, $y = 1$.

We explained already in Section 3 how to reason about linear constraints. (We omitted there the treatment of the disequalities which is routine.) Next, we focus on the reasoning for the multiplication constraint $x \cdot y = z$ in presence of the non-empty ranges $x \in D_x$, $y \in D_y$ and $z \in D_z$. To this end we introduce the following three domain reduction rules:

MULTIPLICATION 1

$$\frac{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y, z \in D_z \rangle}{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y, z \in D_z \cap int(D_x \cdot D_y) \rangle}$$

MULTIPLICATION 2

$$\frac{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y, z \in D_z\rangle}{\langle x \cdot y = z \ ; \ x \in D_x \cap int(D_z/D_y), y \in D_y, z \in D_z\rangle}$$

MULTIPLICATION 3

$$\frac{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y, z \in D_z\rangle}{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y \cap int(D_z/D_x), z \in D_z\rangle}$$

The way we defined the multiplication and the division of the integer intervals ensures that the *MULTIPLICATION* rules *1,2* and *3* are equivalence preserving. Consider for example the *MULTIPLICATION 2* rule. Take some $a \in D_x, b \in D_y$ and $c \in D_z$ such that $a \cdot b = c$. Then $a \in \{x \in \mathcal{Z} \mid \exists z \in D_z \exists y \in D_y \ x \cdot y = z\}$, so $a \in D_z/D_y$ and a fortiori $a \in int(D_z/D_y)$. Consequently $a \in D_x \cap int(D_z/D_y)$. This shows that the *MULTIPLICATION 2* rule is equivalence preserving.

The following example shows an interaction between all three *MULTIPLICATION* rules.

Example 1. Consider the CSP

$$\langle x \cdot y = z \ ; \ x \in [1..20], y \in [9..11], z \in [155..161]\rangle. \tag{4}$$

To facilitate the reading we underline the modified domains. An application of the *MULTIPLICATION 2* rule yields

$$\langle x \cdot y = z \ ; \ x \in \underline{[16..16]}, y \in [9..11], z \in [155..161]\rangle$$

since, as already noted in in Subsection 2.2, $[155..161]/[9..11]) = [16..16]$, and $[1..20] \cap int([16..16]) = [16..16]$. Applying now the *MULTIPLICATION 3* rule we obtain

$$\langle x \cdot y = z \ ; \ x \in [16..16], y \in \underline{[10..10]}, z \in [155..161]\rangle$$

since $[155..161]/[16..16] = [10..10]$ and $[9..11] \cap int([10..10]) = [10..10]$. Next, by the application of the *MULTIPLICATION 1* rule we obtain

$$\langle x \cdot y = z \ ; \ x \in [16..16], y \in [10..10], z \in \underline{[160..160]}\rangle$$

since $[16..16] \cdot [10..10] = [160..160]$ and $[155..161] \cap int([160..160]) = [160..160]$.

So using all three multiplication rules we could solve the CSP (4). □

Now let us clarify why we did not define the division of the sets of integers Z and Y by

$$Z/Y := \{z/y \in \mathcal{Z} \mid y \in Y, z \in Z, y \neq 0\}.$$

The reason is that in that case for any set of integers Z we would have $Z/\{0\} = \emptyset$. Consequently, if we adopted this definition of the division of the integer intervals, the resulting *MULTIPLICATION 2* and *3* rules would not be anymore equivalence preserving. Indeed, consider the CSP

$$\langle x \cdot y = z \ ; \ x \in [-2..1], y \in [0..0], z \in [-8..10]\rangle.$$

Then we would have $[-8..10]/[0..0] = \emptyset$ and consequently by the *MULTIPLICATION 2* rule we could conclude

$$\langle x \cdot y = z \ ; \ x \in \emptyset, y \in [0..0], z \in [-8..10]\rangle.$$

So we reached an inconsistent CSP while the original CSP is consistent.

In the remainder of the paper we will also consider variants of this third approach that allow squaring and exponentiation as atomic constraints. For this purpose we explain the reasoning for the constraint $x = y^n$ in presence of the non-empty ranges $x \in D_x$ and $y \in D_y$, and for $n > 1$. To this end we introduce the following two rules in which to maintain the property that the domains are intervals we use the $int(.)$ operation of Section 2:

EXPONENTIATION

$$\frac{\langle x = y^n \ ; \ x \in D_x, y \in D_y\rangle}{\langle x = y^n \ ; \ x \in D_x \cap int(D_y^n), y \in D_y\rangle}$$

ROOT EXTRACTION

$$\frac{\langle x = y^n \ ; \ x \in D_x, y \in D_y\rangle}{\langle x = y^n \ ; \ x \in D_x, y \in int(D_y \cap \sqrt[n]{D_x})\rangle}$$

To prove that these rules are equivalence preserving suppose that for some $a \in D_x$ and $b \in D_y$ we have $a = b^n$. Then $a \in D_y^n$, so $a \in int(D_y^n)$ and consequently $a \in D_x \cap int(D_y^n)$. Also $b \in \sqrt[n]{D_x}$, so $b \in D_y \cap \sqrt[n]{D_x}$, and consequently $b \in int(D_y \cap \sqrt[n]{D_x})$.

7 Implementation Details

In this section we describe the benchmark experiments that were performed to compare the proposed approaches. These experiments were performed using a single solver of the DICE (DIstributed Constraint Environment) framework. DICE [10] is a framework for solver cooperation, implemented using techniques from coordination programming. It is developed around an experimental constraint solver, called OpenSolver, which is particularly suited for coordination. The coordination and cooperation aspects are irrelevant from the point of view of this paper. Relevant aspects of the OpenSolver are:

– It implements a branch-and-infer tree search algorithm for constraint solving. The inference stage corresponds to constraint propagation and is performed by repeated application of domain reduction functions (DRFs) that correspond to the domain reduction rules associated with the considered constraints.

– This algorithm is abstract in the sense that the actual functionality is determined by software plug-ins in a number of predefined categories. These categories correspond to various aspects of the abstract branch-and-infer tree search algorithm. Relevant categories are: variable domain types, domain reduction functions, schedulers that control the application of the DRFs, branching strategies that split the search tree after constraint propagation has terminated, and several categories corresponding to different aspects of a search strategy that determine how to traverse a search tree.

All experiments were performed using the `IntegerInterval` variable domain type plug-in. Domains of this type consist of an indication of the type of the interval (bounded, unbounded, left/right-bounded, or empty), and a pair of arbitrary precision integer bounds. This plug-in, and the interval arithmetic operations on it are built using the GNU MP library [4].

The branching strategy that we used selects variables using the chronological ordering in which the auxiliary variables come last. The domain of the selected variable is split into two subdomains using bisection, so the resulting search trees are binary trees. In all experiments we searched for all solutions, traversing the entire search tree by means of depth-first leftmost-first chronological backtracking.

For the experiments in this paper a DRF plug-in has been developed that implements the domain reduction rules discussed in the previous sections. The scheduler plug-in used in the benchmarks keeps cycling through the sequence of DRFs, applying DRFs that have been scheduled for execution. When a DRF is applied, and some variable domain is modified, all DRFs that depend on these changes are scheduled for execution, including possibly the one that has just been applied. The cycling stops when no more DRFs are scheduled for execution, or when the domain of a variable becomes empty.

As an alternative to cycling, the scheduler can be supplied with a *schedule*: a sequence of indices into the sequence of DRFs. The scheduler will then cycle through this schedule instead, and consider DRFs for application in the specified order. This is used in combination with the second and third approach, where we distinguish *user* constraints from the constraints that are introduced to define the values of auxiliary variables. Before considering for execution a DRF f that is part of the implementation of a user constraint, we make sure that all auxiliary variables that f relies on are updated. For this purpose, the indices of the DRFs that update these variables precede the index of f in the schedule. If f can change the value of an auxiliary variable, its index is followed by the indices of the DRFs that propagate back these changes to the variables that define the value of this auxiliary variable.

For the third approach, there can be hierarchical dependencies between auxiliary variables. Much like the HC4 algorithm of [2], the schedule specifies a bottom-up traversal of this hierarchy in a forward evaluation phase and a top-down traversal in a backward propagation phase before and after applying a DRF of a user constraint, respectively. In the forward evaluation phase, the DRFs that are executed correspond to rules *MULTIPLICATION 1* and *EXPO-*

NENTIATION. The DRFs of the backward propagation phase correspond to *MULTIPLICATION 2* and *3*, and *ROOT EXTRACTION*. It is easy to construct examples showing that the use of hierarchical schedules can be beneficial compared to cycling through the rules.

The proposed approaches were implemented by first rewriting arithmetic constraints to polynomial constraints, and then to a sequence of DRFs that correspond with the rules of the approach used. We considered the following methods:

1a the first approach, discussed in Section 4,
1b the optimization of the first approach discussed at the end of Section 4 that involves dividing out common powers of variables,
2a the second approach, discussed in Section 5. The conversion to simple polynomial constraints is implemented by introducing an auxiliary variable for every non-linear monomial. This procedure may introduce more auxiliary variables than necessary.
2b an optimized version of approach 2a, where we stop introducing auxiliary variables as soon as the constraints contain no more duplicate occurrences of variables.
3a the third approach, discussed in Section 6, allowing only linear constraints and multiplication as atomic constraints.
3b idem, but also allowing $x = y^2$ as an atomic constraint.
3c idem, allowing $x = y^n$ for all $n > 1$ as an atomic constraint.

Approaches 2 and 3 involve an extra rewrite step, where the auxiliary variables are introduced. The resulting CSP is then rewritten according to approach 1a. During the first rewrite step the hierarchical relations between the auxiliary variables are recorded and the schedules are generated as a part of the second rewrite step. For approaches 2b and 3 the question of which auxiliary variables to introduce is an optimization problem in itself. Some choices result in more auxiliary variables than others. We have not treated this issue as an optimization problem but relied on heuristics. We are confident that these yield a realistic implementation. In our experiments we used the following benchmarks.

Cubes. The problem is to find all natural numbers $n \leq 1000$ that are a sum of four different cubes, for example

$$1^3 + 2^3 + 3^3 + 4^3 = 100.$$

This problem is modeled as follows:

$$\langle 1 \leq x_1,\ x_1 \leq x_2 - 1,\ x_2 \leq x_3 - 1,\ x_3 \leq x_4 - 1,\ x_4 \leq n,\\ x_1^3 + x_2^3 + x_3^3 + x_4^3 = n;\ n \in [1..1000],\ x_1, x_2, x_3, x_4 \in \mathcal{Z}\rangle$$

Opt. We are interested in finding a solution to the constraint $x^3 + y^2 = z^3$ in the integer interval $[1..100000]$ for which the value of $2x \cdot y - z$ is maximal.

Fractions. This problem is taken from [9]: find distinct nonzero digits such that the following equation holds:

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

There is a variable for each letter. The initial domains are [1..9]. To avoid symmetric solutions an ordering is imposed:

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

Also two redundant constraints are added:

$$3\frac{A}{BC} \geq 1 \qquad \text{and} \qquad 3\frac{G}{HI} \leq 1$$

Because division is not present in the arithmetic expressions, the above constraints are multiplied by the denominators of the fractions to obtain arithmetic constraints.

Two representations for this problem were studied:

- *fractions1* in which five constraints are used: one equality and four inequalities for the ordering and the redundant constraints,
- *fractions2*, used in [9], in which three auxiliary variables, BC, EF and HI, are introduced to simplify the arithmetic constraints: $BC = 10B + C$, $EF = 10E + F$, and $HI = 10H + I$.

Additionally, in both representations, 36 disequalities $A \neq B$, $A \neq C$, ..., $H \neq I$ are used.

Kyoto. The problem[1] is to find the number n such that the alphanumeric equation

$$\begin{array}{r} \mathrm{K\,Y\,O\,T\,O} \\ \mathrm{K\,Y\,O\,T\,O} \\ +\,\mathrm{K\,Y\,O\,T\,O} \\ \hline \mathrm{T\,O\,K\,Y\,O} \end{array}$$

has a solution in the base-n number system. Our model uses a variable for each letter and one variable for the base number. The variables K and T may not be zero. There is one large constraint for the addition, 6 disequalities $K \neq Y$ ... $T \neq O$ and four constraints stating that the individual digits K, Y, O, T, are smaller than the base number. To spend some CPU time, we searched base numbers 2..100.

8 Results

Tables 1 and 2 compare the proposed approaches on the problems defined in the previous section. The first two columns of table 1 list the number of variables and

[1] V. Dubrovsky and A. Shvetsov. *Quantum* cyberteaser: May/June 1995, `http://www.nsta.org/quantum/kyotoarc.asp`

Table 1. Statistics and comparison with other solvers

		nvar	nDRF	nodes	activated	%effective	elapsed	E	I
cubes	1a	5	14	167	1880	13.03	0.013	+	=
	2a	9	22	167	2370	22.15	0.014	+	=
	3a	13	34	359	4442	26.23	0.024	-	-
	3b	13	34	227	3759	29.24	0.021	=	-
opt	1a	4	7	115,469	5,186,968	42.16	22.037	+	+
	2a	8	15	115,469	9,799,967	60.00	23.544	+	+
	3a	10	21	?	?	?	?	-	-
	3b	10	21	5,065,137	156,903,869	46.49	518.898	-	-
fractions1	1a	9	154	11,289	1,193,579	3.65	16.586	=	=
	1b	9	154	7,879	734,980	3.45	17.811	=	=
	2a	37	210	11,289	1,410,436	23.27	5.575	=	=
	2b	32	200	11,289	1,385,933	21.65	5.957	=	=
	3	43	208	11,131	1,426,186	27.76	5.635	=	=
fractions2	1a	12	105	2,449	270,833	9.72	0.660	=	=
	1b	12	105	989	94,894	9.12	0.538	=	=
	2a	20	121	2,449	350,380	22.19	0.597	=	=
	2b	15	111	2,449	301,855	17.51	0.547	=	=
	3	22	123	1,525	293,038	27.33	0.509	=	=
kyoto	1a	5	37	87,085	3,299,736	6.09	23.680	=	=
	1b	5	37	87,085	3,288,461	5.94	45.406	+	+
	2a	13	53	87,085	3,781,414	23.03	11.784	=	=
	2b	12	51	87,085	3,622,361	21.45	12.138	=	=
	3a	16	60	87,087	4,275,930	26.70	22.538	=	=
	3b	16	60	87,085	4,275,821	26.70	22.530	=	=
	3c	16	59	87,085	3,746,532	23.26	10.466	=	=

the DRFs that were used. Column nodes lists the size of the search tree, including failures and solutions. The next two columns list the number of times that a DRF was executed, and the percentage of these activations that the domain of a variable was actually modified. For the *opt* problem, the DRF that implements the optimization is not counted, and its activation is not taken into account. The elapsed times in the last column are the minimum times (in seconds) recorded for 5 runs on a 1200 MHz Athlon CPU.

Table 2 lists measured numbers of basic interval operations. Note that for approach 1b, there are two versions of the division and addition operations: one for integer intervals, and one for intervals of reals of which the bounds are rational numbers (marked $\mathcal{Q}$). Columns multI and multF list the numbers of multiplications of two integer intervals, and of an integer interval and an integer factor, respectively. These are different operations in our implementation.

For the *cubes* and *opt* problems, the constraints are already in simple form, so approaches 1a, 1b and 2b are identical. Also all non-linear terms involve either a multiplication or an exponentiation, so also approaches 2a and 3c are the same. The results of these experiments clearly show the disadvantage of implementing exponentiation by means of multiplication: the search space grows because we

Table 2. Measured numbers (thousands) of interval operations

		root	exp	div	multI	multF	sum	total
cubes	1a	1	4	0	0	5	4	14
	2a	< 0.5	< 0.5	0	0	5	4	9
	3a	0	0	1	1	6	5	13
	3b	< 0.5	< 0.5	1	< 0.5	5	5	11
opt	1a	2,299	4,599	1,443	1,444	11,064	5,187	26,037
	2a	1,636	1,538	2,150	738	8,138	4,445	18,645
	3a	?	?	?	?	?	?	?
	3b	21,066	18,105	54,171	18,284	106,651	57,469	275,747
fractions1	1a	0	0	868	28,916	14,238	13,444	57,466
	1b	0	0	51	11,892	8,010	6,727	29,584
				1,550 $\mathcal{Q}$			1,355 $\mathcal{Q}$	
	2a	0	0	734	933	4,736	4,669	11,071
	2b	0	0	776	1,509	5,292	5,147	12,725
	3	0	0	693	339	4,835	4,769	10,636
fractions2	1a	0	0	142	690	304	212	1,347
	1b	0	0	19	127	59	26	344
				65 $\mathcal{Q}$			49 $\mathcal{Q}$	
	2a	0	0	124	149	138	94	505
	2b	0	0	124	206	210	118	658
	3	0	0	114	46	142	101	403
kyoto	1a	735	11,040	1,963	13,852	10,852	13,946	52,388
	1b	735	8,146	218	8,955	12,516	10,592	48,749
				4,310 $\mathcal{Q}$			3,277 $\mathcal{Q}$	
	2a	383	759	1,590	484	5,324	7,504	16,044
	2b	383	759	1,597	1,360	5,756	8,008	17,863
	3a	0	0	1,991	578	5,324	7,505	15,397
	3b	< 0.5	< 0.5	1,990	578	5,324	7,504	15,397
	3c	1	1	1,554	484	5,324	7,504	14,868

increase the number of variable occurrences and lose the information that it is the same number that is being multiplied. For *opt* and approach 3a, the run did not complete within reasonable time and was aborted.

Columns E and I of table 1 compare the propagation achieved by our approaches with two other systems, respectively ECL^{i}PS^e Version 5.6[2] using the ic library, and ILOG Solver 5.1[3] using type ILOINT. For this purpose we ran the test problems without search, and compared the results of constraint propagation. A mark '=' means that the computed domains are the same, '+' that our approach achieved stronger propagation than the solver that we compare with, and '-' that propagation is weaker. For *cubes*, ECL^{i}PS^e computes the same domains as those computed according to approach 3b, so here the reduction is stronger than for 3a, but weaker than for the other approaches. For *opt* ECL^{i}PS^e

[2] ECL^{i}PS^e Constraint Logic Programming System. See `http://www-icparc.doc.ic.ac.uk/eclipse`

[3] See `http://www.ilog.com`

and ILOG Solver compute the same domains. These domains are narrower than those computed according to approaches 3a and 3b, but the other approaches achieve stronger reduction. In all other cases except for *kyoto* and approach 1b the results of all three solvers are the same.

For both representations for the fractions puzzle, the symbolic manipulation of approach 1b is able to achieve a significant reduction of the search tree, but this is not reflected in the timings. For *fractions1* the elapsed time even increases. The reason is that computing the domain updates involves adding intervals of real numbers. The arithmetic operations on such intervals are more expensive than their counterparts on integer intervals, because the bounds have to be maintained as rational numbers. Arithmetic operations on rational numbers are more expensive because they involve the computation of greatest common divisors. For *kyoto* the symbolic manipulation did not reduce the size of the search tree, so the effect is even more severe.

In general, the introduction of auxiliary variables leads to a reduction of the number of interval operations compared to approach 1a. The reason is that auxiliary variables prevent the evaluation of subexpressions that did not change. This effect is strongest for *fractions1*, where the main constraint contains a large number of different power products. Without auxiliary variables all power products are evaluated for every *POLYNOMIAL EQUALITY* rule defined by this constraint, even those power products the variable domains of which did not change. With auxiliary variables the intervals for such unmodified terms are available immediately, which leads to a significant reduction of the number of interval multiplications.

The effect that stronger reduction is achieved as a result of introducing auxiliary variables, mentioned in Section 6, is seen for both representations of the *fractions* benchmark. The effect described in Section 5 is not demonstrated by these experiments.

If we don't consider the symbolic manipulation of approach 1b, approach 3c leads to the smallest total number of interval operations in all cases, but the scheduling mechanism discussed in Section 7 is essential for a consistent good performance. If for example the schedule is omitted for *opt*, the number of interval operations almost triples, and performance of approach 2a and 3c is then much worse than that of approach 1a.

The total numbers of interval operations in table 2 do not fully explain all differences in elapsed times. One of the reasons is that different interval operations have different costs. Especially the preprocessing of the numerator interval for integer interval division, discussed in Subsection 2.2, is potentially expensive, which may explain why for *opt*, approach 1a runs faster than approach 2a, even though the total number of interval operations is higher. Among the many other factors that may be of influence, some overhead is involved in applying a DRF, so if the number of applications differs significantly for two experiments, this probably influences the elapsed times as well (*cubes*, 1a, 2a, *opt*, 1a, 2a, *fractions2*, 2a, 2b). The elapsed times are not the only measure that is subject to implementation details. For example, we implemented division by a constant interval $[-1..-1]$ as multiplication by a constant, which is more efficient in our implementation. Such decisions are reflected in the numbers reported in table 2.

9 Discussion

In this paper we discussed a number of approaches to constraint propagation for arithmetic constraints on integer intervals. To assess them we implemented them using the DICE (DIstributed Constraint Environment) framework of [10], and compared their performance on a number of benchmark problems. We can conclude that:

– Implementation of exponentiation by multiplication gives weak reduction. In our third approach $x = y^n$ should be an atomic constraint.
– The optimization of the first approach, where common powers of variables are divided out, can significantly reduce the size of the search tree, but the resulting reduction steps rely heavily on the division and addition of rational numbers. These operations can be expected to be more expensive than their integer counterparts, because they involve the computation of greatest common divisors.
– Introducing auxiliary variables can be beneficial in two ways: it may strengthen the propagation, as discussed in Sections 5 and 6, and it may prevent the evaluation of subexpressions the variable domains of which did not change.
– As a result, given a proper scheduling of the rules, the second and third approach perform better than the first approach without the optimization, in terms of numbers of interval operations. Actual performance depends on many implementation aspects. However for our test problems the results of variants 2a, 2b and 3c do not differ significantly.

In general, our implementation is slow compared to, for example, ILOG Solver. A likely cause is that we use arbitrary precision integers. We chose this representation to avoid having to deal with overflow, but an additional benefit is that large numbers can be represented exactly.

A different approach would be to use floating-point arithmetic and then round intervals inwards to the largest enclosed integer interval. This was suggested in [3] and implemented in for example RealPaver[4]. A benefit of this inward rounding approach is that all algorithms that were developed for constraints on the reals are immediately available. A disadvantage is that for large numbers no precise representation exists, i.e., the interval defined by two consecutive floating-point numbers contains more than one integer. But it is debatable whether an exact representation is required for such large numbers.

We realize that the current set of test problems is rather limited. In addition to puzzles, some more complex non-linear integer optimization problems should be studied. We plan to further evaluate the proposed approaches on non-linear

[4] `http://www.sciences.univ-nantes.fr/info/perso/permanents/granvil/realpaver/main.html`

integer models for the SAT problem. Also we would like to study the relationship with the local consistency notions that have been defined for constraints on the reals and give a proper characterization of the local consistencies computed by our reduction rules.

Note. This work was performed during the first author's stay at the School of Computing of the National University of Singapore. The work of the second author was supported by NWO, The Netherlands Organization for Scientific Research, under project number 612.069.003.

References

1. K. R. Apt. A proof theoretic view of constraint programming. *Fundamenta Informaticae*, 33(3):263–293, 1998. Available via `http://arXiv.org/archive/cs/`.
2. F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising hull and box consistency. In *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 230–244. The MIT Press, 1999.
3. F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24.
4. T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library, Edition 4.1.2.* Swox AB, December 2002.
5. W. Harvey and P. J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 8(2):173–207, 2003.
6. T. J. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.
7. R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.
8. D. Ratz. Inclusion isotone extended interval arithmetic. Technical report, University of Karlsruhe, 1996. Report No. D-76128 Karlsruhe.
9. C. Schulte and G. Smolka. Finite domain constraint programming in Oz. A tutorial, August 2002. Version 1.2.4 (20020829). Available via
`http://www.mozart-oz.org/documentation/fdt/index.html`.
10. P. Zoeteweij. Coordination-based distributed constraint solving in DICE. In Brajendra Panda, editor, *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 360–366, 2003.

Clustering for Disconnected Solution Sets of Numerical CSPs

Xuan-Ha Vu, Djamila Sam-Haroud, and Boi Faltings

Artificial Intelligence Laboratory,
Swiss Federal Institute of Technology in Lausanne (EPFL),
CH-1015, Lausanne, Switzerland
{xuan-ha.vu,jamila.sam,boi.faltings}@epfl.ch
http://liawww.epfl.ch

Abstract. This paper considers the issue of postprocessing the output of interval-based solvers for further exploitations when solving numerical CSPs with continuum of solutions. Most interval-based solvers cover the solution sets of such problems with a large collection of boxes. This makes it difficult to exploit their results for other purposes than simple querying. For many practical problems, it is highly desirable to run more complex queries on the representations of the solution set. We propose to use clustering techniques to regroup the output in order to provide some characteristics of the solution set. Three new clustering algorithms based on connectedness and their combinations are proposed.

1 Introduction

Many practical applications involve the solving of constraint satisfaction problems (CSPs). A numerical CSP (NCSP) is stated as a set of variables taking their values in domains over the reals and subject to a finite set of constraints. In practice, constraints can be equalities or inequalities of arbitrary type, usually expressed using arithmetic and logical expressions. NCSPs with *non-isolated solutions* are often encountered in real-world engineering applications. In such applications, a set of non-isolated solutions often expresses relevant alternatives that need to be identified as precisely and completely as possible. Interval constraint-based solvers take as input an NCSP and generate a collection of boxes which *conservatively* encloses the solution set. They have shown their ability to solve some complex instances of NCSPs with non-isolated solutions, especially in low-dimensional space. However, they provide enclosures that are still prohibitively verbose for further exploitations rather than just a simple query[1]. More complex queries, on connectedness, satisfaction of universal quantification or intersection for example, are however central to many applications. Although the running time of interval-based solvers is getting gradually improved, the space complexity (the number of boxes) is at least proportional to the number of boxes needed to cover the boundary of the solution set, therefore still very verbose.

[1] In [1], worst-case query time of a *box-tree* in d-dimensions is $\Theta(N^{1-1/d} + k)$, where N is the number of boxes and k is the number of boxes intersecting the *query range*.

One of the applications of interval-based solvers we are investigating is a co-operation scheme of optimization and constraint satisfaction, where the verbose output data of the solvers is usually simplified by imposing a limit on number of boxes to be produced or by restricting the solving process to a low predefined precision. However, in case the solution set is complex and disconnected, the above simplifications affect significantly the quality of the output and make it unsuitable for practical use. We propose to use fast clustering techniques to regroup a large collection of output boxes into a reduced collection of boxes each of which tightly covers one or several connected subsets. The needs for progress on this direction are also ubiquitous in real-world applications such as collision detection, self-collision detection in molecular dynamics and dynamic systems, where the systems have multiple components in low-dimensional space and the reduced collection of boxes is useful for soon guaranteeing that systems have no collisions, hence further expensive computations can be avoided in many cases. We just cite as example some recent works in collision detection ([2], [3], [4], [5], [6], [7], [8]) which have showed that using *bounding-volume techniques* could gain good results. These techniques aim at computing bounding boxes which are very similar to the output from interval-based solvers for NCSPs. In such applications, interval-based solvers can be used to produce *conservative* approximations of the solution sets within acceptable time and clustering techniques can be used to bridge the gap between the verboseness in the output of interval-based solvers and the compactness required by the applications.

It is known that general computation of an optimal set of k clusters is NP-complete [9]. For this reason, fast clusterings usually can be achieved only by using heuristic algorithms or by restricting to approximations. Very recently, a fast clustering algorithm named `ANTCLUST` [10] was proposed. This algorithm is close to, but still not suitable for addressing our requirements. Though successful, the general clustering techniques are not suitable for directly applying to our case because they are not guaranteed to be convergent and the homogeneity information in our problems is not available as required. Therefore, specific clustering techniques for data produced by interval-based solvers are needed.

In Section 3, we first propose a basic clustering algorithm, called `COLONIZATION`, that takes $O(dN^2)$ time to cluster N boxes in d-dimensions. The basic method computes homogeneity (i.e. the connectedness in this case) for each pair of boxes. Moreover, most search techniques implemented in interval-based solvers follow the *branch-and-prune* scheme, hence they essentially produce boxes in the tree structure[2]. Taking advantages of the tree structure, we propose two approximate algorithms to cluster a large collection of boxes organized in form of tree. The first algorithm, called `MCC`, very quickly performs clusterings such that no connected subsets are partitioned. The second algorithm, called `SDC`, generates more adaptive clusterings in very short time. It can be seen as a further process of `MCC`. In Section 3.5, we will also discuss about combining the proposed algorithms to reduce the running time of the basic algorithm.

[2] Even if the tree structure is not available, we still can construct a good bounding-box tree from a list of boxes in $O(N \log N)$ time [1].

2 Background and Definitions

2.1 Interval Arithmetic

The finite nature of computers precludes an exact representation of the *reals*. The real set, $\mathbb{R}$, is in practice approximated by a finite set $\mathbb{F}_\infty = \mathbb{F} \cup \{-\infty, +\infty\}$, where $\mathbb{F}$ is a finite set of reals corresponding to the floating-point numbers. In this paper, we restrict the notion of *interval* to refer to real intervals with bounds in $\mathbb{F}_\infty$ no matter they are open or closed (see [11] for more details). The lower and upper brackets of intervals are taken in $\mathcal{L} = \{``(", ``["\}$ and $\mathcal{U} = \{``)", ``]"\}$ respectively. A total ordering $\prec$ is defined on the set of brackets as follows: $``)" \prec ``[" \prec ``]" \prec ``("$. The set of floating-point bounds is $\mathbb{F}_\diamond = \mathbb{F}_\triangleleft \cup \mathbb{F}_\triangleright$, where $\mathbb{F}_\triangleleft = \mathbb{F} \times \mathcal{L} \cup \{(-\infty, ``(")\}$ and $\mathbb{F}_\triangleright = \mathbb{F} \times \mathcal{U} \cup \{(+\infty, ``)")\}$. $\mathbb{F}_\diamond$ is totally ordered by the ordering $<$ defined as follows: $\forall \beta_1 = (x_1, \alpha_1), \beta_2 = (x_2, \alpha_2) \in \mathbb{F}_\diamond : \beta_1 < \beta_2 \Leftrightarrow x_1 < x_2 \vee (x_1 = x_2 \wedge \alpha_1 \prec \alpha_2)$. The bounds in $\mathbb{F}_\diamond$ are used to construct the set of intervals $\mathbb{I} = \{\langle \beta_1, \beta_2 \rangle \mid \beta_1 < \beta_2, \beta_1 \in \mathbb{F}_\triangleleft, \beta_2 \in \mathbb{F}_\triangleright\}$. We can also use the usual notations of real interval by following the next conventions (where $x, y \in \mathbb{R}$):

$$\begin{aligned}
[x, y] &= \langle (x, ``["), (y, ``]") \rangle &= \{r \in \mathbb{R} \mid x \le r \le y\} \\
(x, y] &= \langle (x, ``("), (y, ``]") \rangle &= \{r \in \mathbb{R} \mid x < r \le y\} \\
[x, y) &= \langle (x, ``["), (y, ``)") \rangle &= \{r \in \mathbb{R} \mid x \le r < y\} \\
(x, y] &= \langle (x, ``("), (y, ``)") \rangle &= \{r \in \mathbb{R} \mid x < r < y\}
\end{aligned}$$

For convenience, for each $\beta = (x, \alpha) \in \mathbb{F}_\diamond$ we define $\neg\beta = (x, \neg\alpha)$, where $\neg``)" = ``[", \neg``[" = ``)", \neg``]" = ``(", \neg``(" = ``]"$.

Interval arithmetic is an arithmetic defined on sets of intervals, rather than sets of real numbers. According to a survey paper [12], a form of interval arithmetic perhaps first appeared in 1924 in [13], then later in [14]. Modern development of interval arithmetic began with Moore's dissertation [15]. Interval arithmetic has been being used to solve numerical problems with guaranteed accuracy. Readers are referred to [16], [17] and [18] for more details on basic interval methods. Most of the interval methods can be easily extended to accept the notation of interval in this paper.

2.2 Relations and Approximations

A relation can be approximated by a computer-representable superset or subset. The former is a *complete* approximation but may contain points that are not solutions. Conversely, the latter is a *sound* approximation but may lose certain solutions. In practice, a relation can be approximated coarsely by the smallest box, called the *(interval) hull* and denoted by *hull*(.), containing it.

A collection of objects representing points is called *disjoint* if every two objects have no common points. Two objects are called *hull-disjoint* if the two hulls of them are disjoint. A set is called a *disconnected set* if it can be partitioned into two nonempty subsets such that each subset has no points in common with the *set closure*[3] of the other, otherwise it is called a *connected set.*

[3] The set closure of a set S is the smallest closed set containing S.

Approximating a relation usually requires a decomposition into simple components such as convex subsets. Checking for disjoint is trivial with boxes but not trivial with general convex sets. Therefore, in this paper we only investigate the decompositions that partition the relation into disjoint boxes.

2.3 Numerical Constraint Satisfaction Problems

A *numerical constraint satisfaction problem* (NCSP) is a CSP, $P = (\mathcal{V}, \mathcal{C}, \mathcal{D})$, where $\mathcal{V}$ consists of d variables taking values in domains $\mathcal{D} = D_1 \times \ldots \times D_d \subseteq \mathbb{R}^d$. In practice, each domain is usually given as one (or several) intervals. A solution to P is a point in $\mathcal{D}$ which satisfies all the constraints in $\mathcal{C}$. In general, the computation of the solution set of an NCSP is intractable. Instead of that, one may consider the computation of inner and outer approximations in form of unions of disjoint boxes, they are therefore called the *inner* and *outer union approximations* [19,20], respectively. Various bisection techniques were described in [18] to compute outer union approximations. Recently, [11] has computed inner union approximations for universally quantified constraints by using the *negation test*, [21] has used the negation test in combination with enhanced splitting strategies to compute outer union approximations for NCSPs. Recently, the work in [19,20] has proposed an improvement on the works in [11,21] to compute inner and/or outer union approximations.

In general, the computation of the union approximations relies on combining the *local consistency* and *global search* techniques to construct a collection of disjoint boxes covering the solution set or contained in it. In numerical domain, the local consistency techniques usually employed are *Box*, *Hull*, *kB*, *Bound* consistency and variants ([22], [23], [24], [25], [26]). Interval-based search techniques usually use the *branch-and-prune* scheme that combines local consistency techniques with orthogonal-splitting strategies, where the *orthogonal-splitting* refers to the split performed in succession by hyper-planes that are orthogonal to axes. In solving NCSPs with non-isolated solutions, local consistency techniques are employed to implement *contracting operators* ([18], [11], [21], [19,20]) in order to narrow the domains during search. In the group of initial search techniques, output can be structured in form of uniform trees like 2^k-tree ([27], [28]) with $k = 3$, or a binary/ternary tree [18], while in another group of recent techniques the output can be structured in form of more general trees ([11], [21], [19], [20]). The number of children of a search node in the latter group does not exceed $2d + 1$ while the number of search nodes in the latter is usually much less than that in the former.

3 Clustering for Disconnected Solution Set

Concepts. When solving NCSPs with non-isolated solutions, interval-based search techniques following the branch-and-prune scheme produce a large collection of boxes that can be maintained in a *bounding-box tree*, where child nodes represent for branchings in search. The bounding boxes stored at search

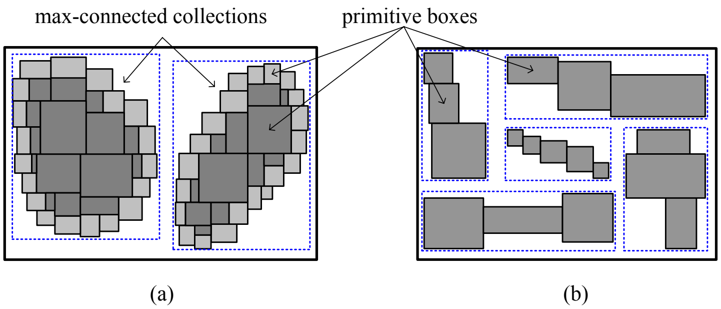


Fig. 1. (a) This tree is orthogonal-separable: grey boxes are primitive boxes, they form two max-connected collections; (b) This tree is not orthogonal-separable

nodes are results of the pruning phase. In literature, there are some variants of bounding-box tree like *bounding-volume tree*, *interval tree*, *box-tree* and *AABB tree*. We call the boxes at the tree leaves that is produced by the solvers the *primitive boxes*. In this paper, we use the notion of *hull of boxes* to refer to the smallest box that contains all the given boxes and also, for convenience, to refer to the collection of the involved boxes if not confusing. A collection of primitive boxes is called *connected* if the union of the boxes is a connected set. If this is the case, we say that the primitive boxes connect to each other. A collection of primitive boxes is called *max-connected* (w.r.t to the set of all primitive boxes) if it is connected and there is no other primitive boxes that connect to the given boxes. A clustering is called *max-connected* if each connected collection in a cluster is max-connected. A bounding-box tree is called *orthogonal-separable* if every decomposition of each bounding-box into pairwise disconnected collections (of primitive boxes) can be performed by sequentially using separating hyper-planes that are orthogonal to axes (see Figure 1).

Goals of Clustering. The solution set of an NCSP with non-isolated solutions usually consists of one or more connected subsets each of which is a continuum. In many applications, there is a need for enclosing the solution set by a collection of disjoint bounding boxes such that each of the connected subsets is contained in only one bounding box. To describe the connected subsets as well as possible, the number of bounding boxes should be as big as possible. Interval-based search techniques usually produce approximations that are either very poor (if being stopped at low precision), or prohibitively verbose (if being stopped at medium/high precision) when solving problems with non-isolated and disconnected solution sets. When a certain quality is required, we obviously need to take the latter setting and then do a clustering on the large collection of boxes. However, the exact solution set is unknown yet, only a collection of primitive boxes is known. The above need is then translated into the need for a max-connected clustering. If a max-connected clustering has the maximum number of clusters, we call it an *optimal max-connected clustering*[4]. The optimal

[4] It is easy to prove that the optimal max-connected clustering exists uniquely.

max-connected clustering of an orthogonal-separable bounding-box tree provides pairwise hull-disjoint clusters each of which is a hull of exact one max-connected collection of primitive boxes (see Figure 13-a). In case the bounding-box tree is not orthogonal-separable, the clusters in the optimal max-connected clustering may not be hull-disjoint (see Figure 13-b). In this case, we may need a further decomposition in order to obtain hull-disjoint, if this property is required by applications (see Figure 13-b).

In the next subsections, we propose three new algorithms and their combinations to address different goals. The first subsection focuses on a basic algorithm that computes the optimal max-connected clustering. It is however not very efficient in practice, we then incrementally propose two alternative algorithms. Each algorithm has two phases, but they have the same first phase. The second subsection describes this common phase. The third subsection describes an algorithm to compute max-connected clusters which are hull-disjoint. The fourth subsection gives an algorithm to compute more adaptive clusterings. In the last subsection, we discuss about combining the proposed algorithms in order to reduce the running time of computing the optimal max-connected clustering.

3.1 Optimal Max-Connected Clustering

As far as we know, there does not exist any algorithm that are suitable to find the optimal max-connected clustering. Fortunately, there exists a recent clustering algorithm named `ANTCLUST` [10] that exploits the phenomenon known as *colonial closure* of ants to create homogeneous groups of individuals. The `ANTCLUST` algorithm addresses the general clustering problem in the way that we can borrow a part for addressing our goals. Inspired by Seidel's *invasion* algorithm [29] and some ideas in the `ANTCLUST` algorithm, we propose a simple deterministic algorithm, called `COLONIZATION`, which is given in Figure 2 to compute the optimal max-connected clustering. This basic algorithm check if each unprocessed box connects to existing collections. The check is taken on each member box of each collection to see if the unprocessed box connect to the collection. The other processes are described in detailed in Figure 2. It is easy to see that in the worst-case the first phase (lines 01-12) takes $d(1+2+\ldots+(N-1)) = dN(N-1)/2$ checks for connectedness of two intervals, where N is the number of primitive boxes in d-dimensions. As a result, the time complexity of the first phase is $O(dN^2)$. At the end of the first phase, all produced collections are max-connected and they have no common points, but their hulls are not guaranteed to be disjoint. The number of collections produced by the first phase is therefore equal to the maximum number, p, of max-connected collections. The second phase (lines 14-16) is optional. It is only for the applications that requires hull-disjoint. The second phase has the time complexity $O(dp^2)$. We obviously have $p \leq N$, hence the total time complexity of the `COLONIZATION` algorithm is $O(dN^2)$, or $O(N^2)$ if d is fixed. In practice, $p \ll N$ and p is bounded for fixed problems.

```
00: algorithm COLONIZATION
01: First phase to obtain optimal max-connectedness:
02:     $\mathcal{L} := \emptyset$;
03:     for each box, $\mathbf{B}$, not in any collection in $\mathcal{L}$ do
04:         $C := \{\mathbf{B}\}$;
05:         for each $C_i \in \mathcal{L}$ do
06:             if $\{\mathbf{B}\} \cup C_i$ is connected then
07:                 $C := C \cup C_i$;
08:                 $\mathcal{L} := \mathcal{L} \setminus \{C_i\}$;
09:             end-if
10:         end-for
11:         $\mathcal{L} := \mathcal{L} \cup \{C\}$;
12:     end-for
13: Optional phase to obtain hull-disjoint:
14:     Replace each collection by the hull of its primitive boxes.
15:     while there exist two hulls that have nonempty intersection do
16:         Combine the two hulls into a single hull.
17:     end-while
18: end
```

Fig. 2. The `COLONIZATION` algorithm

3.2 Separator Computation

In order to cluster the primitive boxes into a number of disjoint hulls, we can use hyper-planes that are orthogonal to axes to separate the primitive boxes. We then define new notations for computing such separating hyper-planes as follows.

Definition 1 (Separator, `SPT`). *Given a hull, $\mathbf{H}$, of primitive boxes, an axis $x \in \mathbb{N}$, and an interval, $I \in \mathbb{I}$. The couple (x, I) is called a* separator *of $\mathbf{H}$ if I is a maximal (w.r.t. the set inclusion) interval that satisfies the following condition for each primitive box, $\mathbf{B}$, in $\mathbf{H}$: $I \subseteq \mathbf{H}|_x \wedge I \cap \mathbf{B}|_x = \emptyset$. When this holds, I is called a* separating interval *and x is called a* separating axis. *The set of all separators of $\mathbf{H}$ on axis x is denoted by $\mathtt{SPT}(\mathbf{H}, x)$. $\mathtt{SPT}(\mathbf{H}) = \cup_x \mathtt{SPT}(\mathbf{H}, x)$.*

Definition 2 (Extension, `EXT`). *Given a hull, $\mathbf{H}$, of primitive boxes, an axis $x \in \mathbb{N}$, and a box $\mathbf{B} \subseteq \mathbf{H}$. A couple (x, I) is called an* extension *of $\mathbf{B}$ w.r.t. $\mathbf{H}$ (on axis x) if I is an interval that is maximal (w.r.t. the set inclusion) in $\mathbf{H}|_x \setminus \mathbf{B}|_x$. When this holds, I is called an* extending interval *and x is called an* extending axis. *We denote by $\mathtt{EXT}(\mathbf{B}, x)$ the set of extensions of $\mathbf{B}$ (w.r.t. $\mathbf{H}$) on axis x.*

Figure 3 gives an illustration of the two above notions. It is easy to see that, on each axis, a hull has at most two extensions w.r.t. its parent hull and that all separators of the hull lie between the two extensions. In a bounding-box tree of primitive boxes, we do a process in a bottom-up manner (e.g. in post-order) to make each bounding box to be the hull of its child bounding boxes. Hence, each node's box becomes the hull of primitive boxes contained in it. A bounding-box

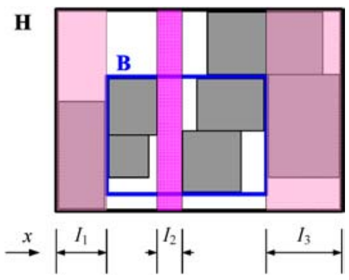


Fig. 3. Grey boxes are primitive; (x, I_2) is a separator of **B** and **H**; (x, I_1) and (x, I_3) are extensions of **B** to **H** on axis x

tree whose bounding boxes are the hulls of primitive boxes contained in the bounding boxes is called a *fitted tree.*

Definition 3 (Separating Set, SE). *In a fitted tree, given an axis $x \in \mathbb{N}$ and a hull,* $\mathbf{H}$*, at a node. The union of the intervals of all separators and extensions*[5] *of* $\mathbf{H}$ *on axis x is called the* separating set *of* $\mathbf{H}$ *on axis x and denoted by* $\mathtt{SE}(\mathbf{H}, x)$.

For simplicity, we use the same notions SPT, EXT, SE for the tree node corresponding to the hull $\mathbf{H}$. The computation of extensions of a hull w.r.t. its parent hull is trivial and takes $O(d)$ time. We can see that, in a fitted tree, a couple (x, I) is a separator of a hull if and only if I is a maximal (w.r.t. the set inclusion) interval that can be contained in all the separating sets on axis x of all children of the hull. It means that all the separators of the hull can be computed from separators and extensions of its children by intersection. Moreover, this is true even during the above-mentioned bottom-up process. The ordering relation among separators and extensions is given by the following proposition.

Proposition 1. *In a fitted tree, given a hull,* $\mathbf{H}$*, of primitive boxes. We have that any two different separating/extending intervals of* $\mathbf{H}$ *on the same axis do not connect to each other. Moreover, for any two separators/extentions, s_1 and s_2, of* $\mathbf{H}$*, either $s_1 = s_2$, $s_1 < s_2$ or $s_2 < s_1$ holds. Where the ordering relation between separators/extentions is defined as follows.*

(i) $(x_1, I_1) = (x_2, I_2) \Leftrightarrow x_1 = x_2 \wedge I_1 = I_2$
(ii) $(x_1, I_1) < (x_2, I_2) \Leftrightarrow x_1 < x_2 \vee (x_1 = x_2 \wedge I_1 < I_2)$

Proof. By Definition 1 and Definition 2, one can easily see that every two different separators/extensions of **H** on the same axis do not connect to each other, otherwise either the separators/extensions are not maximal w.r.t to the set inclusion or **H** is not the hull of its children. As a result, the set of all separating and extending intervals on an axis is totally ordered. This results in what we have to prove.

[5] If not specified, the extensions of a hull are taken w.r.t. the parent hull in the tree.

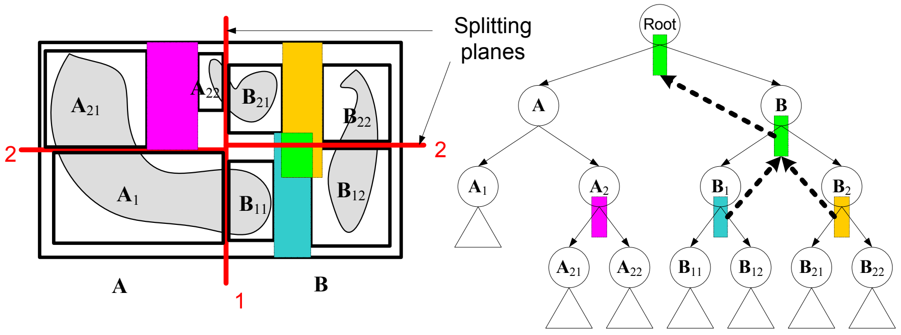


Fig. 4. In the fitting process on a bounding-box tree: the computation of separators is processed in bottom-up manner

To compute separators of all nodes in a bounding-box tree, we only need to do a single bottom-up process, called the *fitting process*. In summary, in the fitting process, operations at each node consist of (i) making the box at the current node to be the hull of its children; and (ii) computing the ordered set of separators by taking the ordered intersection of ordered separating sets of its children. The details of the fitting process is given in Figure 6. We denote by m the maximum number of children of a tree node. The operation (i) is trivial and takes $O(md)$ time. The leaf nodes of the tree have no separators. Each node in the tree has at most two extensions that straddle the box stored at the node, hence straddle the set of separators. Moreover, the bottom-up process starts at leaf nodes, therefore the separating sets can be computed by the intersection in the operation (ii) and maintained sorted by the total order in Proposition 1.

At the current node, we denote by $q_{i,j}$ the number of elements in the separating set on axis $i \in \mathbb{N}$ of the j-th child hull, where $1 \leq i \leq d, 1 \leq j \leq m$. We denote $\bar{q}_j = \sum_i q_{i,j}$ and $\bar{q} = \max_j\{\bar{q}_j\}$, then $\bar{q}$ is the maximum number of separators/extensions that a child of the current node can have. Computing the intersection of a sorted collection of k_1 pairwise disconnected intervals and another sorted collection of k_2 pairwise disconnected intervals takes $O(k_1+k_2)$ time (see Figure 5). By Proposition 1 and the result of *pullSeparators* in Figure 6, $q_{i,j}$ separating/extending intervals in the separating set on axis i of the j-th child are pairwise disconnected and totally ordered. Therefore, the time complexity of computing the intersection of $\sum_j q_{i,j}$ separating/extending intervals on axis i of all children is linear in the number of intervals, i.e. $O(\sum_j q_{i,j})$. As a result, the time complexity of the operation (ii) is $O(\sum_i \sum_j q_{i,j}) = O(\sum_j \sum_i q_{i,j}) = O(\sum_j \bar{q}_j)$, i.e. not greater than $O(m\bar{q})$. Because the number of nodes in the tree is $O(N)$, the total time complexity of the fitting process is $O(mdN + Q)$, where Q is the total number of separators/extensions in the tree except in the root. This can not exceed $O((md+q)N)$, where q is the maximum number of separators/extensions that each node in the tree has. In most existing solvers, m is not greater than $2d+1$, and it is usually small in comparison with $2d+1$ (e.g. $m = 2$ if bisection is used). We conjecture that q is bounded for fixed problems,

```
function $intersect(\{\langle\alpha_i^1,\beta_i^1\rangle \in \mathbb{I} \mid 1 \le i \le k_1\}, \{\langle\alpha_j^2,\beta_j^2\rangle \in \mathbb{I} \mid 1 \le j \le k_2\})$
    $\mathcal{T} := \emptyset$; $i := 1$; $j := 1$;
    while $i \le k_1 \wedge j \le k_2$ do
        if $\neg\beta_i^1 \le \alpha_j^2$ then $\{i := i + 1$; continue while;$\}$
        if $\neg\beta_j^2 \le \alpha_i^1$ then $\{j := j + 1$; continue while;$\}$
        $\alpha := \max\{\alpha_i^1, \alpha_j^2\}$; $\beta := \min\{\beta_i^1, \beta_j^2\}$;
        $\mathcal{T} := \mathcal{T} + \{\langle\alpha,\beta\rangle\}$;
        if $\beta = \beta_i^1$ then $i := i + 1$;
        if $\beta = \beta_j^2$ then $j := j + 1$;
    end-while
    return $\mathcal{T}$;
end
```

Fig. 5. This function computes the intersection of two ordered collections of pairwise disconnected intervals. The intersection of multiple collections is computed by calling this function

```
function $FittingProcess$(in/out : $\mathcal{T}_0$)        /* $\mathcal{T}_0$ is a bounding-box tree */
    for each node $\mathbf{P}$ of $\mathcal{T}_0$ in a post-order visit do
        if $\mathbf{P}$ is leaf then
            for each axis $x$ do SPT($\mathbf{P}, x$) := $\emptyset$;
        else
            $pullSeparators(\mathbf{P})$;
        end-if
    end-for
end

function $pullSeparators$(in/out : $\mathbf{P}$)        /* $\mathbf{P}$ is a tree node */
    if $\mathbf{P}$ is leaf then return;        /* Needed for the calls in Figure 9 */
    $\mathcal{C} := children(\mathbf{P})$;
    Set the box at $\mathbf{P}$ to $hull(\{\mathbf{B} \in \mathbb{I}^d \mid \mathbf{B}$ is a box at a node in $\mathcal{C}\})$.
    for each $\mathbf{C} \in \mathcal{C}$ and each axis $x$ do
        Compute EXT($\mathbf{C}, x$) and denote it by $\{E_l, E_u\}$.        /* may be empty */
        SE($\mathbf{C}, x$) := $\{E_l$, SPT($\mathbf{C}, x$), $E_u\}$;        /* ordered set */
    end-for
    for each axis $x$ do
        SPT($\mathbf{P}, x$) := $intersect\{$SE($\mathbf{C}, x$) $\mid \mathbf{C} \in \mathcal{C}\}$; /* Ignore $x$ to get the intervals */
    SPT($\mathbf{P}$) := {SPT($\mathbf{P}$, 1), ..., SPT($\mathbf{P}, d$)};        /* ordered set */
end
```

Fig. 6. The functions for the fitting process

particularly it is bounded by $O(p)$, where p is given in Section 3.1. If this conjecture is true, as our experiments show, the time complexity of the fitting process is $O(N)$ for fixed problems.

3.3 Max-Connected Clustering

The second phase is called the *separating process*. We now compute a max-connected clustering based on next propositions. During the separating process, we maintain max-connected collections of primitive boxes in form of fitted trees.

Proposition 2. *Any separator stored at the root of a fitted tree representing a max-connected collection can be used to partition this tree into two fitted subtrees each of which represents a max-connected collection.*

Proposition 3. *If a set of primitive boxes represented by a fitted tree can be partitioned by a hyper-plane that is orthogonal to axis x into two collections whose projections on x do not connect to each other, then the root of the fitted tree has some separating interval that contains the projection of the hyper-plane on x.*

Proof of these propositions is trivial due to Definition 1 and the definition of fitted tree. By these propositions, we can compute a max-connected clustering from the separators computed in Section 3.2. The process can be done in bottom-up manner, or simply by a recursive call at root. For simplicity, we describe the process in recursive mode. A recursive call starts from a root node of a fitted tree created during the separating process. Recursively, for each node, $\mathbf{P}$, and the current separator, $S \in \mathtt{SPT}(\mathbf{P})$, we construct two fitted trees from the subtree rooted at $\mathbf{P}$. One tree is called *lower-bound tree* (denoted by $l_\mathbf{P}$), the other is *upper-bound tree* (denoted by $u_\mathbf{P}$) with single roots being copied from $\mathbf{P}$ at first. If a child node's box lies on lower-bound (respectively upper-bound) side of S, the subtree rooted at the child node is moved to $l_\mathbf{P}$ ($u_\mathbf{P}$, respectively). Otherwise the child node, called $\mathbf{M}$, is processed similarly to construct its lower-bound and upper-bound trees, $l_\mathbf{M}$ and $u_\mathbf{M}$ respectively. The trees $l_\mathbf{M}$ and $u_\mathbf{M}$ are then attached to $l_\mathbf{P}$ and $u_\mathbf{P}$ respectively. If any tree root in this process has only one child, this child is shifted to the root. Additionally, we make roots of lower-bound and upper-bound trees to be hulls of their child nodes. The separators of the root nodes of new lower- and upper-bound trees are updated from its children.

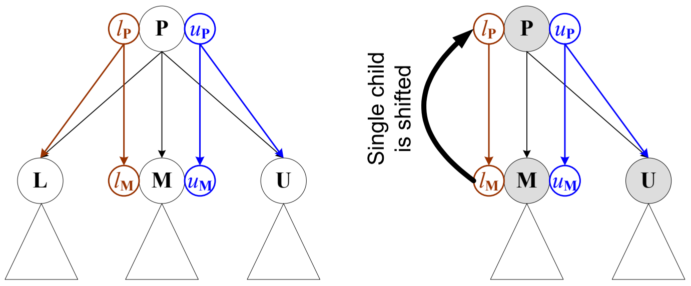


Fig. 7. The separating process: construct lower- and upper-bound trees from the subtrees on lower side and upper side respectively of a separator, and from lower- and upper-bound trees of the children that cut the separator

```
/* $\mathcal{T}_0$ is the initial bounding-box tree, $\mathcal{L}$ is a list of fitted tree to be return */
algorithm MCC(in : $\mathcal{T}_0$) $\rightarrow$ out : $\mathcal{L}$
    $FittingProcess(\mathcal{T}_0)$; $\mathcal{L} := \{\mathcal{T}_0\}$;
    while $\exists \mathcal{T} \in \mathcal{L}$: $\mathcal{T}$ has at least one separator, $S$, at root do
        $\mathcal{L} := (\mathcal{L} \setminus \mathcal{T}) \cup separateSubtree(root(\mathcal{T}), S)$;     /* In Figure 9 */
    end-while
end
```

Fig. 8. The Max-Connected Clustering (MCC) algorithm

```
/* Input: $\mathbf{P}$ is a node of a fitted tree, separator $S \in$ SPT($\mathbf{P}$) */
function $separateSubtree$(in : $\mathbf{P}, S$) $\rightarrow$ out : $\{l_\mathbf{P}, u_\mathbf{P}\}$
    Remove the separator $S$ from SPT($\mathbf{P}$).
    Create two trees, $l_\mathbf{P}$ and $u_\mathbf{P}$, each has a single node copied from $\mathbf{P}$.
    for each $\mathbf{C} \in children(\mathbf{P})$ do
        if $\mathbf{C}$ lies on lower side of $S$ then
            Move the subtree rooted at $\mathbf{C}$ to a new subtree of the root of $l_\mathbf{P}$.
        else if $\mathbf{C}$ lies on upper side of $S$ then
            Move the subtree rooted at $\mathbf{C}$ to a new subtree of the root of $u_\mathbf{P}$.
        else
            Find the separator $S' \in$ SPT($\mathbf{C}$) : $S \subseteq S'$.
            $\{l_\mathbf{C}, u_\mathbf{C}\} := separateSubtree(\mathbf{C}, S')$;
            Attach $l_\mathbf{C}$ (resp. $u_\mathbf{C}$) as a new subtree of the root of $l_\mathbf{P}$ ($u_\mathbf{P}$ resp.).
        end-if
    end-for
    If any root in trees $l_\mathbf{P}$ or $u_\mathbf{P}$ has only one child, shift this child to the root.
    $pullSeparators(root(l_\mathbf{P}))$; $pullSeparators(root(u_\mathbf{P}))$;
end
```

Fig. 9. The separating process of a node/subtree

In Figure 8, we describe the *Max-Connected Clustering* (MCC) algorithm that uses the separating process to get a hull-disjoint max-connected clustering. For simplicity, in the MCC algorithm we use only one list, $\mathcal{L}$, however in implementation it should be maintained as two lists: one list for the trees that have no separators at root and the other list for the rest. In Figure 9, we describe the details of the separating process for a subtree rooted at a node in a fitted tree. Figure 7 and Figure 10 give some illustrations of the separating process. Figure 13-b gives another example on which MCC provides the optimal max-connected clustering. Without difficult, Based on Proposition 3 we can prove that the obtained clustering is optimal max-connected if the bounding-box tree is orthogonal-separable. As proved in Section 3.2, the time complexity of the fitting process is $O(mdN + Q)$. There are at most Q separators in the trees, each separator is processed once, each process takes $O(m)$ time. Therefore, the time complexity of the separating process is $O(mQ)$. The total time complexity of MCC is hence $O(mdN + mQ)$. This is practically shown to be linear in N for fixed problems.

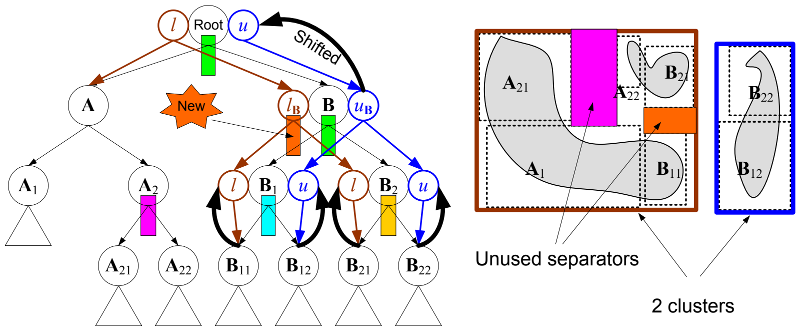


Fig. 10. MCC: The separators exist in the root of the fitted tree are used for recursive separating the tree into lower- and upper-bound subtrees. The figure on the right gives the result of this process. Two obtained clusters are hull-disjoint and max-connected

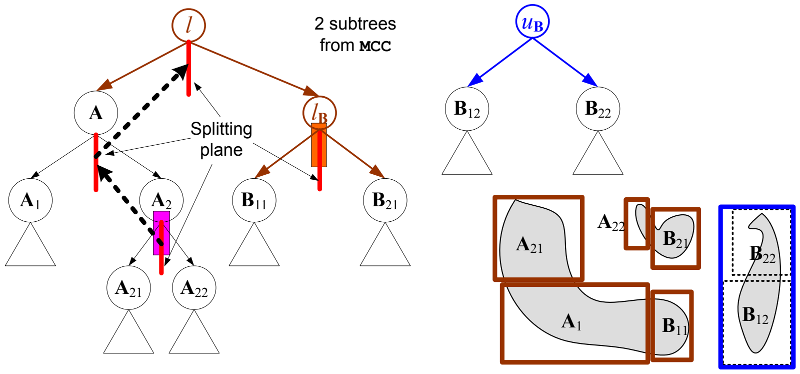


Fig. 11. SDC: The separators that still exist in subtrees after running MCC are considered as hints for decomposition. The process produces six clusters (boxes) that are pairwise hull-disjoint, but not all are max-connected

3.4 Separator-Driven Clustering

In some applications, the clustering obtained by the MCC algorithm characterizes the solution set not well enough (see the example in Figure 10). A remedy for this problem is given as an additional process for MCC so that applications can choose to whether to run it. We observe that the separators that still exist in output fitted trees of the MCC algorithm (e.g. the two separators on the right side in Figure 10) can be used as hints for further separations.

If we consider the remaining separators in the output fitted trees of MCC as hints for further decomposition, we will need to use all the splitting hyper-planes (i.e. the branchings in the trees) in the paths from the tree nodes of those separators upward to the roots of the trees for the separation. That is, all siblings of the nodes from the current node upward to the root are to be separated into different groups/collections in the clustering. The use of splitting hyper-planes

```
/* $\mathcal{T}_0$ is the initial bounding-box tree, $\mathcal{L}$ is a list of fitted tree to be return */
algorithm SDC(in : $\mathcal{T}_0$) $\to$ out : $\mathcal{L}$
    $\mathcal{L} := \emptyset$; $\mathcal{L}_0 :=$ MCC($\mathcal{T}_0$);    /* call to the MCC algorithm */
    for each fitted tree $\mathcal{T} \in \mathcal{L}_0$ do
        $\mathcal{L} := \mathcal{L} \cup separateByAllSeparators(\mathcal{T})$;
    end-for
end

function $separateByAllSeparators$(in : $\mathcal{T}$) $\to$ out : $\mathcal{L}$
    Search in post-order, from left to right, for a node $\mathbf{N}_0$ such that SPT($\mathbf{N}_0$) $\neq \emptyset$.
    if not found then return $\mathcal{L} := \{\mathcal{T}\}$;
    $\mathcal{L} := \emptyset$; $\mathbf{N} := \mathbf{N}_0$; $\mathbf{P} := parent(\mathbf{N})$;    /* $\mathbf{P}$ may be null */
    while $\mathbf{P} \neq \emptyset$ do
        for each $\mathbf{C} \in children(\mathbf{P})$ do
            if $\mathbf{C}$ on the left of $\mathbf{N}$ then Move the subtree rooted at $\mathbf{C}$ to $\mathcal{L}$;
            if $\mathbf{C}$ on the right of $\mathbf{N}$ then
                Detach the subtree rooted at $\mathbf{C}$ and make a new tree $\mathcal{T}_\mathbf{C}$.
                $\mathcal{L} := \mathcal{L} \cup separateByAllSeparators(\mathcal{T}_\mathbf{C})$;
            end-if
            if $\mathbf{C} = \mathbf{N} \neq \mathbf{N}_0$ then Erase the node $\mathbf{C}$;
            if $\mathbf{C} = \mathbf{N} = \mathbf{N}_0$ then
                Detach the subtree rooted at $\mathbf{C}$ and make a new tree $\mathcal{T}_\mathbf{C}$.
                Find a separator $S \in$ SPT($\mathbf{C}$);
                $\mathcal{L} := \mathcal{L} \cup separateSubtree(root(\mathcal{T}_\mathbf{C}), S)$;    /* In Figure 9 */
            end-if
        end-for
        $\mathbf{N} := \mathbf{P}$; $\mathbf{P} := parent(\mathbf{N})$;    /* $\mathbf{P}$ may be null */
    end-while
end
```

Fig. 12. The Separator-Driven Clustering (SDC) algorithm

for the separation does not guarantee that the clusters are pairwise disconnected. In Figure 12, we describe the main steps of an algorithm, called *Separator-Driven Clustering* (SDC), which performs the above idea as an additional process for the MCC algorithm. It is obvious that this algorithm produces a hull-disjoint clustering. Figure 11 gives the result of the further process for the subtrees in Figure 10. Six clusters (boxes) obtained in Figure 11 describe the solution set better than two boxes in Figure 10. In Figure 13-b, we give another example on the SDC algorithm where the solution set consisting of two connected subsets is well covered by 13 boxes each of which is a hull of primitive boxes. The SDC algorithm quickly provides an adaptive clustering as shown in our experiments. Following an argument similar to the argument in Section 3.3, we have the time complexity of SDC is $O(mdN + mQ)$. This is also practically shown to be linear in N for fixed problems.

Proposition 4. *Each cluster produced by* SDC *is a connected set.*

Proof. If there are two primitive boxes in the subtree of a cluster that form a disconnected set, then there must be a separator at their common ancestor. This contradicts the property of SDC: no separators exist in output subtrees.

3.5 Combinations of Algorithms

Three proposed algorithms produce the same results (See Figure 13-a) for orthogonal-separable bounding-box trees. However, for the bounding-box tree that is not orthogonal-separable (See Figure 10, Figure 11, Figure 13-b and Figure 14), the results of the proposed algorithms are different. Only COLONIZATION and MCC guarantee the max-connectedness of the produced clusters. COLONIZATION allows to get the best max-connected clustering but the time complexity is quadratic. MCC only allows to get a hull-disjoint max-connected clustering, it however has a very short running time in our experiments. SDC is the most adaptive clustering in very short running time, but it does not guarantee the max-connectedness. Therefore, we need to investigate the combinations of the proposed algorithms.

***Combination of* MCC *and* COLONIZATION.** As mentioned in Section 3.3, MCC provides a max-connected clustering such that the clusters are pairwise hull-disjoint, hence, if we apply COLONIZATION to every fitted tree produced by MCC, we will get the optimal max-connected clustering. The combined algorithm is therefore called the *Optimal Max-Connected Clustering* (OMCC) algorithm. Let N_i $(1 \leq i \leq p')$ is the number of primitive boxes in the i-th fitted tree produced by MCC, where p' $(\leq p)$ is the number of subtrees produced by MCC. The running time of COLONIZATION on each tree is $O(dN_i^2)$ $(1 \leq i \leq p')$. Hence, the total running time of OMCC is $time(\texttt{MCC}) + O(d\sum_i N_i^2) = O(mdN + mQ + d\sum_i N_i^2)$. Noting that $\sum_i N_i = N$, we have $\sum_i N_i^2 \leq N^2$, therefore $O(mdN + mQ + d\sum_i N_i^2)$ does not exceed $O(dN^2)$. In practice we often see that Q is much smaller than $O(dN^2)$, and $\sum_i N_i^2 \ll N^2$, hence the actual running time of OMCC is better than that of COLONIZATION for problems with highly disconnected solution set, i.e. when p' is big.

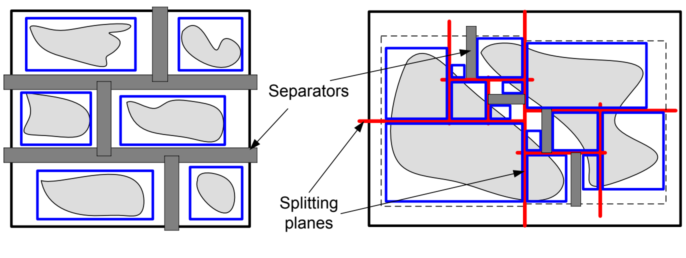


Fig. 13. The bounding-box tree is (a) orthogonal-separable: all algorithms produce the same result; (b) not orthogonal-separable: COLONIZATION produces 2 clusters (or 1 box if perform the optional phase), MCC produces 1 box, SDC produces 13 boxes

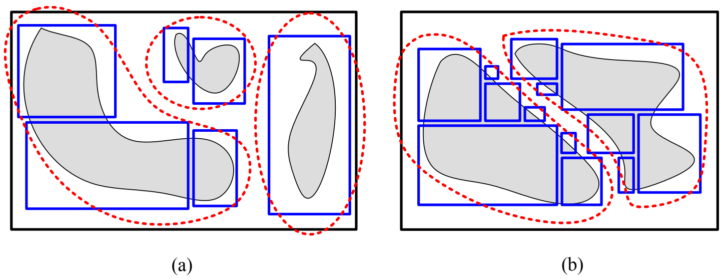


Fig. 14. (a) Applying `OSDC` to the problem in Figure 11 to get 3 clusters; (b) Applying `OSDC` to the problem in Figure 13-b to get 2 clusters

Combination of `SDC` ***and*** `COLONIZATION`. When p' is small, `OMCC` will not be efficient. Instead of running `COLONIZATION` after `MCC`, we continue `MCC` until the end of `SDC` to get n pairwise hull-disjoint clusters. Each of these clusters is a connected set by Proposition 4. Therefore, if we apply `COLONIZATION` to the hull boxes of these clusters, we will get a *near* optimal max-connected clustering. This combined algorithm is then called *Optimized Separator-Driven Clustering* (`OSDC`), that combines the adaptiveness of `SDC` and the optimal max-connectedness of `COLONIZATION`. The complexity of the second phase of `OSDC` (i.e. the use of `COLONIZATION`) is $O(dn^2)$. The total running time of `OSDC` is $time(\texttt{SDC}) + O(dn^2) = O(mdN + mQ + dn^2)$. Note that $p' \leq p \leq n$. In practice of NCSPs, n is bounded for fixed problems, and if p' is very small then n is small, therefore the running time of `OSDC` is close to the running time of `SDC`.

4 Experiments

We now present an evaluation on 16 non-linear problems which have at most four dimensions and which are selected to reflect different topologies of solution set. The selected problems are categorized into three groups: (i) problems with only one connected subset; (ii) problems with several connected subsets; and (iii) problems with tens to hundreds connected subsets. Our experiments show the similarity in the running time of each algorithm in each group. Therefore, we only need to give the average running time and the average number of computed clusters in individual groups on the left and right of the cells, respectively, in Table 1. Figure 15 shows the graph of the average running time.

The results show that the running time of `COLONIZATION` is quadratic in N (the number of boxes) while the running times of `MCC` and `SDC` are very short (the average time of clustering 5000 boxes is less than 50ms, and the running time is always less than 120ms). The running time of the solver that uses the search algorithm in [20] is linear in N. For all problems, the running time of `MCC` and `SDC` much less than that of `COLONIZATION`, and close to zero. In all the tested problems, `COLONIZATION` and `MCC` provide the same clusterings. For most tested problems, the three algorithms produce the same clusterings, though `SDC` is better than the others in the following cases. They only show significant differences

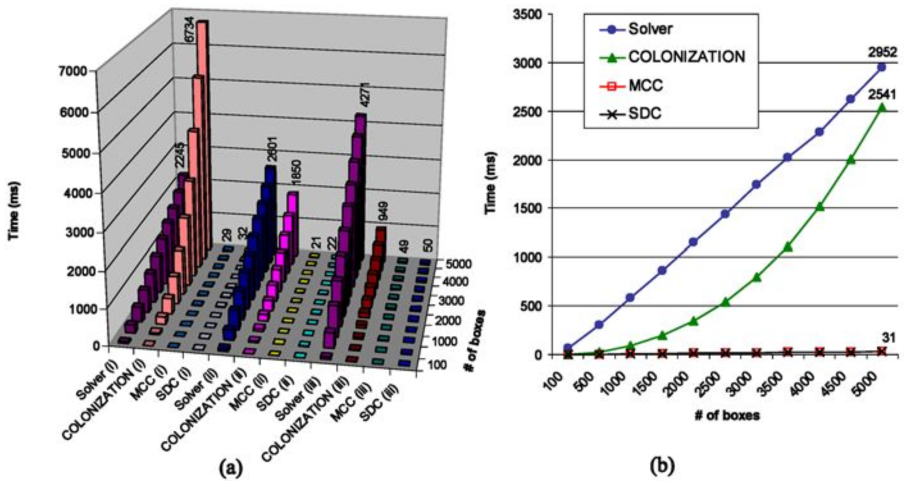


Fig. 15. The average running times in milliseconds (a) in three groups; (b) for all problems

Table 1. The average running times in milliseconds (on the left of cells) and the average number of clusters (on the right of cells) in three groups

$N\ \rightarrow$	100	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
(i) Solver	41	205	409	619	823	1044	1285	1519	1701	1952	2245
(i) `COLONZ.`	2|1.0	48|1.0	195|1.0	430|1.0	779|1.0	1251|1.0	1937|1.0	2762|1.0	3985|1.0	5328|1.0	6734|1.0
(i) `MCC`	1|1.0	2|1.0	5|1.0	8|1.0	10|1.0	14|1.0	16|1.0	19|1.0	21|1.0	26|1.0	29|1.0
(i) `SDC`	1|3.3	2|3.3	6|3.3	9|3.3	12|3.3	15|3.3	17|3.3	20|3.3	23|3.3	28|3.3	32|3.3
(ii) Solver	69	285	525	774	1024	1263	1538	1788	2002	2288	2601
(ii) `COLONZ.`	2|3.1	25|3.6	78|3.7	172|3.9	290|4.0	447|4.1	636|4.1	863|4.4	1120|4.4	1453|4.4	1850|4.4
(ii) `MCC`	0|3.1	2|3.6	4|3.7	6|3.9	8|4.0	10|4.1	12|4.1	15|4.4	16|4.4	19|4.4	21|4.4
(ii) `SDC`	0|5.3	2|11.8	4|11.3	7|11.2	9|10.4	11|10.3	13|11.0	15|9.7	17|9.0	19|9.0	22|9.0
(iii) Solver	66	415	860	1253	1683	2112	2557	2935	3371	3883	4271
(iii) `COLONZ.`	1|9.0	19|24.5	46|31.5	83|47.3	148|69.8	229|97.3	310|116.8	420|116.8	579|116.8	740|116.8	949|116.8
(iii) `MCC`	1|9.0	6|24.5	12|31.5	18|47.3	24|69.8	31|97.3	38|116.8	41|116.8	42|116.8	46|116.8	49|116.8
(iii) `SDC`	1|26.8	7|79.5	12|109.5	18|118.5	24|131.3	32|136.5	39|116.8	41|116.8	43|116.8	46|116.8	50|116.8

in the following problems: $F2.4 = \{\sin(x \sin y) \geq \cos(y \cos x); -4 \leq x, y \leq 4\}$; $G1.2 = \{x_1^2 + 0.5x_2 + 2(x_3 - 3) \geq 0, x_1^2 + x_2^2 + x_3^2 \leq 25, \forall i : -8 \leq x_i \leq 8\}$; $H1.1 = \{x_1^2 + x_2^2 + x_3^2 \leq 9, (x_1 - 0.5)^2 + (x_2 - 1)^2 + x_3^2 \geq 4, x_1^2 + (x_2 - 0.2)^2 \geq x_3, \forall i : -4 \leq x_i \leq 4\}$. For example, when $N = 1000$, `COLONIZATION` and `MCC` provide six boxes for the problem $F2.4$, the volume ratio of the six boxes to the hull of primitive boxes is 0.761 while the ratio obtained by `MCC` is 0.385 with 30 boxes. When $N = 1000$ for problem $G1.2$ (and $H1.1$ respectively), `COLONIZATION` and `MCC` produce no reduction while `SDC` produces 4 (5, respectively) boxes with the volume ratio is 0.225 (0.191, respectively). The experiments show that even in case `COLONIZATION` and `MCC` cannot characterize the solution set well (e.g. the problems $G1.2$ and $H1.1$), `SDC` still can provide an adaptive clustering that are more suitable for applications. We have only done experiments with three fundamental algorithms (`COLONIZATION`, `MCC` and `SDC`), however the performance of `OMCC` and `OSDC` could be deduced from the performance of `MCC` and `SDC` respectively, and the performance of `COLONIZATION` for a small number of boxes (less than 120 boxes in our experiments, then takes less than 2-3ms).

5 Conclusion

Three algorithms and their combinations, that address different goals, have been proposed to regroup the verbose output of interval-based solvers into a fewer number of clusters. In our experiments, `SDC` shows to be the best among the techniques we proposed and tested. Moreover, `MCC` and `SDC` are very cheap (in running time) postprocessing techniques to get useful grouping information for further process. We also can deduce that, for applications that require the optimal max-connected clustering, either `OSDC` or `OMCC` could be alternative to `COLONIZATION` in case the number of clusters is small or big, respectively. We are investigating a cooperation of optimization and numerical constraint satisfaction that could exploit these fast postprocessing techniques to make the interval-based solvers useful in the context of optimization. Potentially, applying these postprocessing techniques to the output of interval-based solvers makes it possible to use the solvers in various applications (as mentioned in Section 1) that require concise representations of the solution set.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This research is funded by the European Commission and the Swiss Federal Education and Science Office through the COCONUT project (IST-2000-26063).

References

1. Agarwal, P., de Berg, M., Gudmundsson, J., Hammar, M., Haverkort, H.: Box-Trees and R-Trees with Near-Optimal Query Time. In: Proceedings of the 17th ACM Symposium on Computational Geometry, ACM Press (2001) 124–133
2. van den Bergen, G.: Efficient Collision Detection of Complex Deformable Models using AABB Trees. Journal of Graphics Tools **4(2)** (1997) 7–25
3. Fahn, C.S., Wang, J.L.: Efficient Time-Interrupted and Time-Continuous Collision Detection Among Polyhedral Objects in Arbitrary Motion. Journal of Information Science and Engineering **15** (1999) 769–799
4. Zhou, Y., Suri, S.: Analysis of a Bounding Box Heuristic for Object Intersection. In: Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA'99). (1999)
5. Ganovelli, F., Dingliana, J., O'Sullivan, C.: BucketTree: Improving Collision Detection Between Deformable Objects. In: Spring Conference on Computer Graphics (SCCG'2000). (2000)
6. Larsson, T., Akenine-Moller, T.: Collection Detection for Continuously Deforming Bodies. In: Eurographics'2001, Manchester, UK (2001)
7. Haverkort, H., de Berg, M., Gudmundsson, J.: Box-Trees for Collision Checking in Industrial Installations. In: Proceedings of the 18th ACM Symposium on Computational Geometry, ACM Press (2002) 53–62
8. Fahn, C.S., Wang, J.L.: Adaptive Space Decomposition for Fast Visualization of Soft Object. Journal of Visualization and Computer Animation **14(1)** (2003) 1–19

9. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York (1979)
10. Labroche, N., Monmarché, N., Venturini, G.: A New Clustering Algorithm Based on the Chemical Recognition System of Ants. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'2002), France, IOS Press (2002) 345–349
11. Benhamou, F., Goualard, F.: Universally Quantified Interval Constraints. In: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000). (2000) 67–82
12. Kearfott, B.: Interval Computations: Introduction, Uses, and Resources. Euromath Bulletin **2(1)** (1996) 95–112
13. Burkill, J.: Functions of Intervals. Proceedings of the London Mathematical Society **22** (1924) 375–446
14. Sunaga, T.: Theory of an Interval Algebra and its Applications to Numerical Analysis. RAAG Memoirs **2** (1958) 29–46
15. Moore, R.: Interval Arithmetic and Automatic Error Analysis in Digital Computing. PhD thesis, Stanford University, USA (1962)
16. Moore, R.: Interval Analysis. Prentice Hall, Englewood Cliffs, NJ (1966)
17. Hickey, T., Ju, Q., Van Emden, M.: Interval Arithmetic: from Principles to Implementation. Journal of the ACM (JACM) **48(5)** (2001) 1038–1068
18. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. First edn. Springer (2001) ISBN 1-85233-219-0.
19. Vu, X.H., Sam-Haroud, D., Silaghi, M.C.: Approximation techniques for non-linear problems with continuum of solutions. In: Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation (SARA 2002). Volume 2371 of LNAI., Canada, Springer-Verlag (2002) 224–241
20. Vu, X.H., Sam-Haroud, D., Silaghi, M.C.: Numerical Constraint Satisfaction Problems with Non-isolated Solutions. In: Global Optimization and Constraint Satisfaction: First International Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS 2002), France, October 2-4, 2002, Revised Selected Papers. Volume 2861 of LNCS., Springer-Verlag (2003) 194–210
21. Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Search Techniques for Non-linear CSPs with Inequalities. In: Proceedings of the 14th Canadian Conference on AI. (2001)
22. Lhomme, O.: Consistency Techniques for Numeric CSPs. In: Proceedings of IJCAI'93. (1993)
23. Van Hentenryck, P.: A Gentle Introduction to NUMERICA. Artificial Intelligence **103** (1998) 209–235
24. Collavizza, H., Delobel, F., Rueher, M.: A Note on Partial Consistencies over Continuous Domains. In: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP'98). (1998)
25. Granvilliers, L., Goualard, F., Benhamou, F.: Box Consistency through Weak Box Consistency. In: Proceedings of ICTAI'99. (1999)
26. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising Hull and Box Consistency. In: Proceedings of the International Conference on Logic Programming (ICLP'99), Las Cruces, USA (1999) 230–244
27. Sam-Haroud, D., Faltings, B.: Consistency Techniques for Continuous Constraints. Constraints **1(1-2)** (1996) 85–118
28. Lottaz, C.: Collaborative Design using Solution Spaces. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland (2000)
29. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press (1993)

Implementing Propagators for Tabular Constraints

Roman Barták and Roman Mecl

Charles University in Prague, Faculty of Mathematics and Physics
Institute for Theoretical Computer Science
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
{bartak,mecl}@kti.mff.cuni.cz

Abstract. Many real-life constraints describing relations between the problem variables have complex semantics. It means that the constraint domain is defined using a table of compatible tuples rather than using a formula. In the paper we study the implementation of filtering algorithms (propagators) for such constraints that we call tabular constraints. In particular, we propose compact representations of extensionally defined binary constraints and we describe filtering algorithms for them. We concentrate on the implementation aspects of these algorithms so the proposed propagators can be naturally integrated into existing constraint satisfaction packages like SICStus Prolog.

Introduction

Many real-life problems can be naturally modeled as a constraint satisfaction problem (CSP) using variables with a set of possible values (domain) and constraints restricting the allowed combinations of values. Quite often, the semantics of the constraint is well defined via mathematical and logical formulas like comparison or implication. However, the intentional description of some constraints is rather complicated and it is more convenient to describe them extensionally as a set of compatible tuples. A relation between the type of activity and its duration or the allowed transitions between the activities are typical examples of such constraints in the scheduling applications [1,2]. The constraint domain is specified there as a table of compatible tuples rather than as a formula, thus we are speaking about tabular constraints. Figure 1 shows an example of such a tabular constraint.

X	Y	
1	2..20, 30..50	
2	-	*No compatible value*
3	inf..sup	*No restriction on Y*
4	2..20, 30..50	

Fig. 1. Example of a tabular constraint: a range of compatible values of Y is specified for each value of X.

* Research is supported by the Grant Agency of the Czech Republic under the contract no. 201/01/0942.

In this paper we study the filtering algorithms (propagators) for binary constraints where the constraint domain is described as a table. We propose two new filtering algorithms that use a compact representation of the constraint domain. Such a compact representation turned out to be crucial for the efficiency of algorithms applied to real problems with non-trivial domains containing thousands or millions of values. Rather than providing a full theoretical study of the algorithms we concentrate on the practical aspects of the implementation, which are usually omitted in research papers. In particular, the algorithms are proposed in such a way that they can be easily integrated into existing constraint solvers.

The paper is organized as follows. First, we give a motivation for using tabular constraints and we survey existing approaches to model such constraints. Then we describe two new filtering algorithms for tabular constraints. These algorithms extend our previous works on tabular constraints [3,4] by including better entailment detection and by using a more compact representation of the constraint domain. We also propose the algorithms for converting tables to the compact representation. We conclude the paper by an empirical study of the algorithms.

Motivation

Our work on filtering algorithms for tabular constraints is motivated by practical problems where important real-life constraints are defined in the form of tables. In particular, this work is motivated by complex planning and scheduling problems where the user states constraints over the objects like activities and resources. In complex environments, there could appear rather complicated relations between the activities expressing, for example, transitions between two activities allocated to the same resource like changing a color of the produced item [1]. Typically, the activities are grouped in such a way that the transitions between arbitrary two activities within the group are allowed but a special set-up activity is required for the transition between two activities of different groups. The most natural way to express such a relation is using a table describing the allowed transitions (Figure 2).

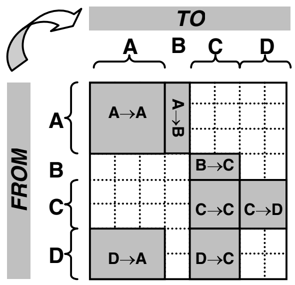


Fig. 2. A transition constraint expressed as a table of compatible transitions (shadow regions) between eight activities grouped into four groups A, B, C, and D.

As we showed in [2], there are many other constraints of the above type in real-life planning and scheduling problems, for example a description of the time windows, duration, and the cost of the activity. The users often define such constraints in the form of a table describing the set of compatible pairs. Therefore, we are speaking about *tabular constraints*. Because it is rather complicated to convert such a table into a mathematical formula defining a constraint with efficient propagation, it is more convenient to use special filtering algorithms handling the tabular constraints directly. Efficiency of such filtering algorithms can be improved by exploiting information about the typical structure of the constraint domain in a given application. For example, we have noticed that the structure of many above mentioned binary tabular constraints consists of several possible overlapping rectangles (see Figure 2) and the number of such rectangles is much smaller than the number of compatible pairs.

Related Works

Arc consistency (AC) or, in other words, propagation over binary constraints is studied for a long time and many AC algorithms have been proposed. Since 1986, we have the AC-4 algorithm [16] with an optimal worst-case time complexity. The average time complexity of this algorithm has been improved in its followers like AC-6 [6] and AC-7 [7]. All these algorithms are fine grained in the sense that they are working with individual pairs of compatible values – they use so called value based constraint graphs. Moreover, these algorithms need "heavy" data structures to minimize the number of consistency checks. These features complicate implementation and make the algorithms impractical due to a space complexity when large domains of variables are used. Therefore, an older but a simpler algorithm AC-3 [14] is used more widely in the constraint packages like SICStus Prolog, ECLiPSe, ILOG Solver, CHIP, Mozart etc. Actually, a variant of this algorithm that is called AC-8 [12] is used by these systems. AC-8 uses a list of variables with the changed domain instead of the list of constraints to be revised that is used by AC-3.

Recently, the new algorithms AC-3.1 [20] and AC-2000/2001 [8] based on the AC-3 schema have been proposed. AC-3.1 and AC-2001 achieve an optimal worst-case time complexity without using heavy data structures. However, they still require the data structures for individual values in the variables' domains which could complicate their usage for very large domains due to a space complexity.

In this paper, we concentrate on filtering algorithms for extensionally defined binary constraints over large domains. The proposed filtering algorithms are intended for existing constraint solvers so these algorithms must fit in the AC-3 (AC-8) schema. To achieve a good time and space complexity of the algorithms, we are exploiting the structure of the constraint domain. There exist several works about AC algorithms exploiting the structure of the constraint domain. For example, the generic AC-5 algorithm [19] achieves better time efficiency for functional, monotonic, and anti-functional constraints. The paper [11] describes a technique for converting the extensionally represented constraints into a set of simple constraints.

The existing constraint solvers usually provide a mechanism to model extensionally defined constraints without necessity to program a new filtering algorithm. For example, the `element` constraint is often included in the constraint solvers. *N* such `element` constraints can model arbitrary *N*-ary constraint. However, because every consistent tuple must be specified there, it is not possible to represent the constraints with infinite domains like the constraint from Figure 1.

In SICStus Prolog [17], there is a `relation` constraint where the user may specify a binary constraint as a list of compatible pairs similar to the table from Figure 1. In particular, for each value of the first (leading) variable, the user describes a range of compatible values of the second (dependent) variable. The range is a finite set of disjoint intervals. The domain for the leading variable must be finite and till the version 3.8.7 the range for the dependent variable must consist of finite intervals only.

The latest versions of SICStus Prolog (since 3.10.0) provide a generalization of the `relation` constraint. This new constraint called `case` allows compact modeling of arbitrary N-ary relations similar to our models. We compare empirically our filtering algorithms both to `relation` and `case` constraints later in the paper. Unfortunately, the filtering algorithms behind the `relation` and `case` constraints are not published which prevents a deeper comparison of the techniques.

In [4] a straightforward filtering algorithm called general relation was proposed. This algorithm supports infinite domains for the dependent variable and it provides a mechanism to detect constraint entailment when the reduced constraint domain has a rectangular structure [3]. Later in the paper we describe an efficient extension to this algorithm that uses a more compact representation of the constraint domain.

In [5], a new technique called sweep was proposed to explore constraint domains. This technique was applied to tabular constraints in [3]. The sweep filtering algorithm represents the constraint domain using a rectilinear rectangular covering – a set of possibly overlapping rectangles – so the representation is more compact. However, this algorithm has no mechanism to detect constraint entailment. Later in the paper, we present an extension to this algorithm that includes a detector of constraint entailment and that uses a more compact representation of the constraint domain.

Preliminaries

Constraint programming is a framework for declarative problem solving by stating constraints over the problem variables and then finding a value for each variable in such a way that all the constraints are satisfied. The value for a particular variable can be chosen only from the variable domain that is from a set of possible values for the variable. *Constraint* is an arbitrary relation restricting the possible combinations of values for the constrained variables. The *constraint domain* is a set of value tuples satisfying the constraint. For example, {(0,2), (1,1), (2,0)} is a domain of the constraint X+Y=2 where variables' domains consist of non-negative integers. If C is a constraint over the ordered set of variables Xs then we denote the constraint domain C(Xs). We say that the constraint domain has a *rectangular structure* if $C(Xs) = \times_{X\in Xs} C(Xs)\downarrow X$, where $C(Xs)\downarrow X$ is a projection of the constraint domain to the variable X (Figure 3). For example, the above constraint X+Y=2 does not have a rectangular structure because the projection to both variables is {0,1,2} and the Cartesian product {0,1,2}×{0,1,2} is larger than the constraint domain. The notion of a rectangular structure is derived from the domain structure of binary constraints where the constraint domain forms a rectangle with possible vertical and horizontal gaps.

Assume that C(Xs) is a domain of the constraint C and D(X) is a domain of the variable X – a set of values. We call the intersection $C(Xs) \cap (\times_{X\in Xs} D(X))$ a *reduced domain* of the constraint. Note, that the reduced domain consists only of the tuples $(v_1,\ldots,v_n)$ such that $\forall i\ v_i\in D(X_i)$.

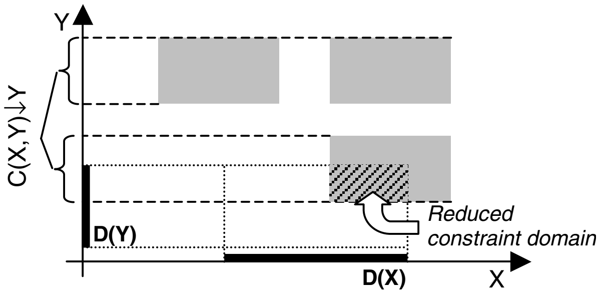


Fig. 3. Example of a constraint domain (shadow rectangles), its projection to the variable Y (C(X,Y)↓Y), and a reduced constraint domain (the striped rectangle).

Many constraint solvers are based on maintaining consistency of constraints during enumeration of variables. We say that the constraint is *consistent* (arc-consistent, hyper arc-consistent)[1] if every value of every variable participating in the constraint is part of some assignment satisfying the constraint. More precisely, every value of every variable participating in the constraint must be part of some tuple from the reduced constraint domain. For example, the constraint X+Y=2, where both the variables X and Y have domain {0,1,2}, is consistent while the constraint from Figure 3 is not consistent. To make the constraint consistent we can reduce the domains of involved variables by projecting the reduced constraint domain to the variables:

$$\forall Y \in Xs\text{: } D(Y) \leftarrow (C(Xs) \cap (\times_{X \in Xs} D(X)))\downarrow Y.$$

The algorithm that makes the constraint consistent is called a *propagator* [10]. More precisely, the propagator is a function that takes variables' domains as the input and that proposes a narrowing of these domains as the output. The propagator is *complete* if it makes the constraint consistent that is all locally incompatible values are removed. The propagator is *sound* if it does not remove any value that can be part of the solution. The propagator is *idempotent* if it reaches a fix point that is the next application of the propagator to the narrowed domains does not narrow them more.

We say that the constraint satisfaction problem is (hyper) arc-consistent, if every constraint is consistent. It is not enough to make every constraint consistent by a single call to its complete propagator because the domain change might influence consistency of already consistent constraints. Thus the propagators are called repeatedly in a propagation loop until there is no domain change. In fact, the particular propagator is called only when the domain of any variable involved in the constraint is changed. Many constraint systems allow a finer definition when the propagator should be evoked via so called *suspensions*, for details see [9,10]. Nevertheless, the existing constraint solvers rarely go beyond the arc-consistency schema in the propagation loop.

When the domains of all variables in the constraint contain exactly one value then it is not necessary to call the propagator again because it will not narrow the domains anymore. However, the propagator may be stopped even sooner. Assume that the

[1] The notion of arc-consistency is used for binary constraints only. For constraints of higher arity, the notions of hyper arc-consistency or generalised arc-consistency are used. For simplicity reasons we will use the term consistency there.

domain of X is {1,2,3} and the domain of Y is {5,6,7}. Then the propagator for the constraint X<Y deduces no domain narrowing. This is because every combination of values from the variables' domains satisfies the constraints - the constraint is entailed. We say that the constraint is *entailed* if the constraint is satisfied for any combination of values from variables' domains. Visibly, the constraint is entailed if and only if the reduced constraint domain has a rectangular structure.

The rest of the paper deals with the propagators for extensionally defined binary constraints over totally ordered domains. We expect the propagator to be evoked when the domain of any variable involved in the constraint is changed. Our goal is to design efficient, complete, idempotent, and sound propagators.

Compact General Relation

The general relation (GR) constraint or more precisely the GR propagator was first described in [4]. This propagator uses a set representation of the constraint domain where one variable is selected as the leading variable and the other variable is dependent. The constraint domain is represented as a set of pairs (x,dy), where x is a value of the leading variable and dy is a set of compatible values of the dependent variables (Figure 4). The values of the leading variable are pair-wise different. This is a natural representation of the constraints that are described using a table like in Figure 1. This representation requires a finite projection of the constraint domain to the leading variable and a finitely representable projection to the dependent variable, for example a finite set of disjoint intervals that we call a *range*. Notice that this representation also covers some infinite constraint domains.

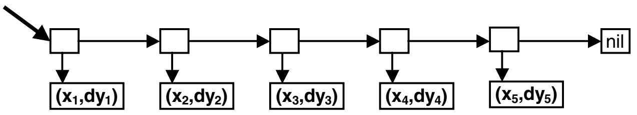


Fig. 4. Representation of the constraint domain by the GR propagator.

The filtering algorithm proposed in [4] simply explores the set representing the constraint domain and tests whether x_i is a part of the current domain of X and whether the intersection of dy_i with the current domain of Y is non-empty. In such a case x_i remains in the domain of X and $dy_i \cap$ D(Y) will be a part of the narrowed domain of Y.

When using the above algorithm with real-life constraints in a scheduling application [2], we have noticed that many dy_i are identical. Thus, we can compact the domain representation to reduce memory consumption and to speed-up the filtering algorithm.

Domain Generator

Let T = $\{(x_i,dy_i) \mid i=1..n\}$ be a representation of the binary constraint domain where x_i are pair-wise different values of the leading variable and dy_i is a range of values of the dependent variable that are compatible with the value x_i. Such representation can be derived directly from the table defining the constraint. Because the time complexity of the above sketched GR propagator depends on the size of T, it could be beneficial to compact the representation and to upgrade the GR propagator for such a compact representation. In particular, it is possible to compact all pairs (x_i,dy_i) with identical dy_i component. Formally, for the original set T we get a new compacted set:

$$CT = \{(dx_i,dy_i) \mid dx_i = \{ x \mid (x,dy_i)\in T \} \ \& \ dx_i \neq \emptyset \}$$

We use a straightforward algorithm that converts T into CT with the time complexity $O(n.log\ n)$, where n is a number of the elements in the set T. First, the algorithm orders lexicographically the set T according to dy_iThen, the pairs (dx_i,dy_i) with the identical component dy_i form continuous sub-sequences in the ordered set and thus it is easy to collect them in a linear time. Figure 5 shows an example of such a compact representation.

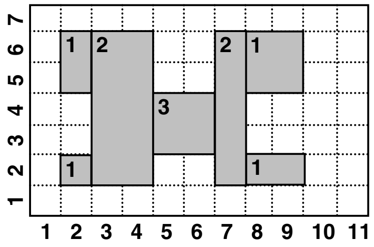


Fig. 5. A decomposition of the constraint domain (shadow rectangles) into a set of non-overlapping sub-domains with the rectangular structure.

Notice that the pair (dx,dy) in CT describes an area with a rectangular structure in the constraint domain and all these areas are pair-wise disjoint. Actually, the constraint domain is decomposed into a set of areas with a rectangular structure. This simplifies the filtering algorithm that can handle each such area independently as we will show in the next section. In general, the proposed filtering algorithm requires the areas to have a rectangular structure but it does not require them to be disjoint. Thus, we can see there a possibility to design other decompositions of the constraint domain that are perhaps even more compact.

On the other hand, the proposed filtering algorithm includes an entailment detector that requires the *dx* components to be disjoint (CT has this feature). Consequently, the areas are disjoint as well. Thus, if this particular entailment detector is used (and we will show later that it brings some speed-up) then CT is the optimal decomposition[2]. The open question is whether it is possible to design efficient entailment detectors that do not require the above feature.

Filtering Algorithm

The filtering algorithm for the compacted GR relation mimics the behavior of the original filtering algorithm from [4] that we described above. There are just few

[2] CT has the smallest number of rectangular areas such that their union equals to the constraint domain and their projections to the leading variable are disjoint.

changes to respect the new compact representation of the constraint domain. Figure 6 describes the new compact GR propagator.

The compact GR propagator incrementally constructs the projection of the reduced constraint domain to both variables by exploring the areas in the compact representation of the constraint domain. For each area `(DX,DY)`, the propagator checks whether the area has a non-empty intersection with the reduced constraint domain (rows 10-13). Actually, the propagator constructs a *reduced area* `(CompatibleX,CompatibleY)` and the projections of this reduced area to both variables become parts of the narrowed domains of the variables (rows 17-18, 30-31).

```
1  procedure GR(Constraint,X,Y)
2    NewDomainOfX ← empty
3    NewDomainOfY ← empty
4    ConstraintDomain ← domain(Constraint)
5    Entailed ← true
6    LastProjectionOfY ← empty
7    NewDomain ← empty
8    while non_empty(ConstraintDomain) do                     Domain
9      (DX,DY) ← head(ConstraintDomain)                       filtering
10     CompatibleX ← intersection(domain(X),DX)
11     if non_empty(CompatibleX) then
12       CompatibleY ← intersection(domain(Y),DY)
13       if non_empty(CompatibleY) then
14         if empty(NewDomain) then
15           NewDomain ← ConstraintDomain                      Domain
16         end if                                              shift
17         NewDomainOfX ← union(NewDomainOfX, CompatibleX)
18         NewDomainOfY ← union(NewDomainOfY, CompatibleY)
19         if Entailed then                                    Entailment
20           if empty(LastProjectionOfY) then                  detector
21             LastProjectionOfY ← CompatibleY
22           else
23             Entailed ← (LastProjectionOfY == CompatibleY)
24           end if
25         end if
26       end if
27     end if
28     ConstraintDomain ← tail(ConstraintDomain)
29   end while
30   X in NewDomainOfX
31   Y in NewDomainOfY
32   domain(Constraint) ← NewDomain
33 end GR
```

Fig. 6. The filtering algorithm of the compact GR propagator.

The propagator might change the representation of the constraint domain to keep only the reduced constraint domain. This would help when the propagator is called next time because a smaller number of smaller areas will be explored which would speed-up the propagator. However, many constraint solvers including the solvers in Prolog keep the domains in memory after any change to allow fast recovery of the domain upon backtracking [10]. The paper [4] showed that it significantly increases memory consumption for a simple GR propagator that updates the constraint domain. So instead of keeping the reduced constraint domain, a technique called *domain shift* has been proposed in [4] to keep only a part of the reduced constraint domain. The set modeling the constraint domain is represented as a list there and domain shift means skipping the areas at the beginning of the list that are not part of the reduced constraint domain. This reduces a bit the size of the constraint domain (the number of areas to be explored when the propagator is called next time) while keeping low memory consumption. The compact GR propagator uses the same technique (rows 14-16, 32).

Last but not least, we have accompanied the compact GR propagator by an *entailment detector*. The entailment detector checks whether the reduced constraint domain has a rectangular structure. Because the projections of the areas to the leading variable are disjoint (this is a feature of CT), the entailment detection is done simply by comparing the projections of the non-empty reduced areas to the dependent variable (rows 19-24). If all these projections are identical then the constraint is entailed so it is not necessary to evoke the propagator again because it will not deduce any domain pruning. Note, that the research papers usually omit the implementation details like entailment detection. However, entailment detection may improve the time efficiency as we will show later.

Time complexity of the compact GR propagator depends on the number of areas in the representation of a constraint domain. Each time the propagator is evoked, every such area is explored (rows 8-29) and thus having a smaller number of areas in the domain representation is an advantage. That is the reason why the compact GR propagator is more time and space efficient then the original GR propagator.

Sweep Filtering Algorithm

The GR propagator uses a straightforward decomposition of the constraint domain to non-overlapping areas with a rectangular structure. Moreover, the projections of these areas to the leading variable are disjoint (Figure 5) which has no effect on the filtering algorithm but it simplifies detection of constraint entailment. In [15] a different decomposition of the constraint is proposed, in particular the decomposition into a set of rectangles covering the constraint domain, so called rectilinear rectangular covering [18]. The filtering algorithm for such decomposition is based on the technique called sweep that is widely used in computational geometry and that was first applied to domain filtering in [5]. The sweep algorithm moves a vertical line called a sweep line along the axis of the leading variable. Each time it encounters or leaves a rectangle – this is called an event – it triggers some event handler according to the event type. Thus the algorithm sweeps the plane, hence its name.

In this paper, we propose a generalization of the filtering algorithm from [3]. It is based on observation that the sweep filtering algorithm can use more general objects than simple rectangles. The algorithm requires the object to have a rectangular structure and its projection to the leading variable to be an interval. We call such an object a *generalized rectangle* (Figure 7). In the next section, we will present a new algorithm constructing the domain representation with generalized rectangles from the original table modeling the constraint domain.

Domain Generator

Before the constraint domain can be used by a sweep pruning algorithm, it must be first decomposed into a set of generalized rectangles. It is easy to get a sequence of non-overlapping generalized rectangles from the original set $T = \{(x_i,dy_i) \mid i=1..n\}$

describing the constraint domain. Simply, the neighboring (in the sense of values of the leading variable) pairs with the identical *dy* component are joined so we get a set:

$$\mathrm{CT}_{\mathrm{sweep}} = \{(min_i \,..\, max_i,\ dy_i) \mid min_i \leq max_i \ \&\ \forall x\ min_i \leq x \leq max_i\!: (x,dy_i)\in \mathrm{T}\}.$$

Notice the difference from the compact GR model; now the projection of an object in $\mathrm{CT}_{\mathrm{sweep}}$ to the leading variable is an interval and thus $|\mathrm{CT}|\leq|\mathrm{CT}_{\mathrm{sweep}}|$. Because the efficiency of the filtering algorithm depends on the number of generalized rectangles, we decided to generate a more compact decomposition from $\mathrm{CT}_{\mathrm{sweep}}$. The decomposition algorithm simply joins the neighboring parts of the rectangles. The idea is as follows: the algorithm takes the generalized rectangle and it tries to extend it to the largest possible *x*. Then this generalized rectangle is removed from the constraint domain and the process is repeated until the domain is empty (Figure 8).

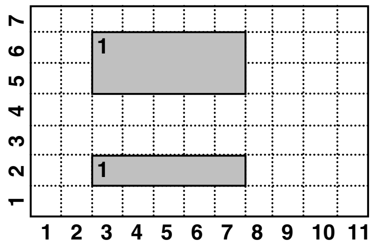


Fig. 7. Example of a generalized rectangle. This rectangle can be represented using the term `rect(3,7,[2,5..6])`.

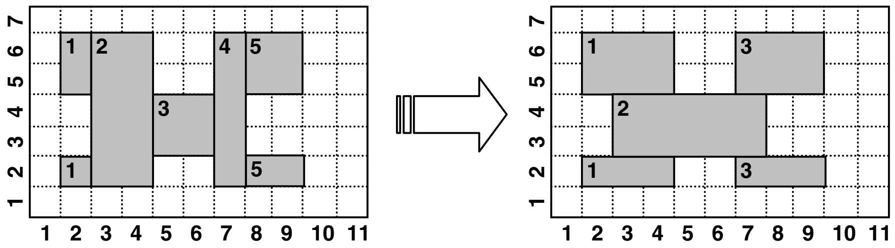


Fig. 8. The number of generalized rectangles covering the constraint domain can be decreased by using a different decomposition of the constraint domain.

We present here a decomposition algorithm based on the sweep technique (Figure 9). The set $\mathrm{CT}_{\mathrm{sweep}}$ is ordered increasingly in the values min_i. Then, the decomposition algorithm explores the generalized rectangles from the ordered set $\mathrm{CT}_{\mathrm{sweep}}$ and it tries to extend each rectangle to the right. To do this job, the algorithm keeps a set of the rectangles that can be extended, so called active rectangles (*ActiveRects*), as well as an "active" projection of these rectangles to the dependent variable (*ActiveDy*). The projections of the currently active rectangles to the dependent variable are disjoint. Thus, if an active rectangle is closed (see below) then we can simply remove its projection from the "active" projection (row 12). Each time the algorithm takes a new rectangle, it tests whether the active rectangles can still be extended to this new rectangle (row 9). If an active rectangle cannot be extended then it is removed from the set of active rectangles and it is put to the final decomposition (*Rects*) - we call it closing the rectangle (rows 12-13). After extending all the active rectangles, the remaining part of the new rectangle (if any) will be included among the active rectangles (rows 17-20). When all the rectangles are explored then the remaining active rectangles are closed (rows 23-25).

```
1    procedure GenerateRectangles(D)
2     Rects ← empty
3     ActiveRects ← empty
4     ActiveDy ← empty
5     LastX ← inf
6     for each (Xmin..Xmax,Dy) in D (in increasing order of Xmin) do
7       TmpRects ← empty
8       for each r(RXmin,RDy) in ActiveRects do               Rectangle
9         if RDy ⊆ Dy && LastX+1=Xmin then                    extension
10            TmpRects ← r(RXmin,RDy) : TmpRects
11        else
12            ActiveDy ← ActiveDy - RDy
13            Rects ← rect(RXmin,LastX,RDy) : Rects
14        end if
15      end for
16      ActiveRects ← TmpRects
17      if non_empty(Dy - ActiveDy) then
18        ActiveRects ← r(Xmin, Dy - ActiveDy) : ActiveRects        New
19        ActiveDy ← Dy                                        rectangle
20      end if
21      LastX ← Xmax
22    end for
23    for each r(Xmin,Dy) in ActiveRects do
24      Rects ← rect(Xmin,LastX,Dy) : Rects
25    end for
26   end GenerateRectangles
```

Fig. 9. The algorithm for domain decomposition.

In the worst case, the number of rectangles generated by the above algorithm will be $|CT_{sweep}|$. However, the algorithm decreases the number of rectangles in many cases (see Figure 8). Note also that the presented decomposition algorithm generates non-overlapping rectangles. Figure 10 demonstrates the run of the algorithm.

Rectangles	*LastX*	*ActiveDy*	*ActiveRects*	*Rects*
	inf	empty	empty	empty
(2..2)-[2,5..6]	2	[2,5..6]	r(2,[2,5..6])	empty
(3..4)-[2..6]	4	[2..6]	r(3,[3..4]), r(2,[2,5..6])	empty
(5..6)-[3..4]	6	[3..4]	r(3,[3..4])	rect(2,4,[2,5..6])
(7..7)-[2..6]	7	[2..6]	r(7,[2,5..6]), r(3,[3..4])	rect(2,4,[2,5..6])
(8..9)-[2,5..6]	9	[2,5..6]	r(7,[2,5..6])	rect(3,7,[3..4]), rect(2,4,[2,5..6])
				rect(7,9,[2,5..6]), rect(3,7,[3..4]), rect(2,4,[2,5..6])

Fig. 10. Example run of `GenerateRectangles` with the constraint domain from Figure 8.

Filtering Algorithm

The filtering algorithm based on the sweep technique was proposed in [15, 3] and the same sweep pruning algorithm can be used for generalized rectangles without any modification. The sweep pruning (SP) algorithm moves a vertical line called a *sweep line* along the horizontal axis from left to right. Each time it encounters or leaves an object – this is called an *event* – it triggers a relevant event handler. In case of domain filtering, there are four types of events used by the sweep algorithm:

rect_start(PosX,NumR,IntY) - indicates the left border (*PosX*) of the rectangle identified by *NumR* with the vertical projection *IntY*,

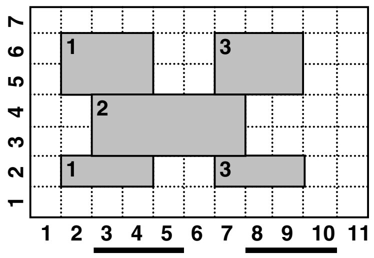
rect_end(PosX,NumR) - indicates the right border (*PosX*) of the rectangle identified by *NumR*,

x_start(PosX) - indicates the start of some continuous interval within the current domain of the leading variable,

x_end(PosX) - indicates the end of some continuous interval within the current domain of the leading variable.

The list of events can be generated in advance from the constraint domain and from the current domain of the leading variable. We call such a list an *event point series*. The events in the event point series are ordered increasingly according to the x-coordinate of the event (*PosX*). Moreover, we require the start events to precede the end events with the same x-coordinate. This is necessary for the algorithm to capture "one-point" overlaps between the objects. Figure 11 shows an example of the event point series for the constraint domain consisting of three generalized rectangles and the domain of the leading variable consisting of two intervals.

The SP algorithm incrementally builds the new domains for both variables by exploring the generalized rectangles (Figure 12). *ListOfX* keeps a list of bounds of the intervals in the new domain of the leading variable (in the reverse order). This list is then converted to the domain of the dependent variable (rows 10-15). *ListOfDomY* is a list of projections of the rectangles, which have non-empty intersection with the reduced constraint domain, to the dependent variable.



Event point series:
rect_start(2,1,[2,5..6]),
rect_start(3,2,[3..4]),
x_start(3),
rect_end(4,1),
x_end(5),
rect_start(7,3,[2,5..6),
rect_end(7,2),
x_start(8),
rect_end(9,3),
x_end(10)

Fig. 11. A constraint domain (left) and its corresponding event point series (right).

During the computation, the SP algorithm keeps some global data structures that describe the status of computation:

InDomain: indicates whether the sweep line is within the domain of the leading variable that is between *x_start* and *x_end* events corresponding to a single continuous interval,

ActiveRects: describes the set of rectangles that are currently crossed by the sweep line that is the rectangles where the *rect_start* event has been processed and the corresponding *rect_end* has not been reached yet.

We have added a simple entailment detector (row 18) to the SP algorithm. If all the rectangle projections in *ListOfDomY* are identical then the constraint is entailed. Visibly, this entailment detector is not complete because it does not detect all constraint entailments. For example, assume the constraint domain from Figure 11, the domain of the leading variable to be {4,7}, and the domain of the dependent variable to be (3..5). Then the constraint is entailed but the algorithm does not detect it because the projections of the rectangles 1 and 2 are not identical.

```
1   procedure SP(Constraint,X,Y)
2     EventPointSeries ← make_event_point_series(Constraint,X)
3     ListOfDomY, ActiveRects, ListOfX ← empty
4     DY ← domain(Y)
5     InDomain ← false
6     while non_empty(EventPointSeries) do
7       Event ← select_and_delete_first(EventPointSeries)
8       process_event(Event,DY,ActiveRects,InDomain,ListOfX,ListOfDomY)
9     end while
10    NewDomainOfX ← empty
11    while non_empty(ListOfX) do
12      Max ← select_and_delete_last(ListOfX)
13      Min ← select_and_delete_last(ListOfX)
14      NewDomainOfX ← union(Min..Max,NewDomainOfX)
15    end while
16    X in NewDomainOfX
17    Y in intersection(union(ListOfDomY),DY)
18    Entailed ← all elements in ListOfDomY are identical
19  end SP
```

Fig. 12. The filtering algorithm of the SP propagator.

The power behind the SP algorithm is hidden in the procedures for event processing (Figure 13). Notice that only the rectangles having a non-empty projection to the domain of the dependent variable are processed (rows 20, 29). Let us call these *rectangles relevant*.

If the sweep line enters a relevant rectangle (*rec_start* event) and it is within the domain of the leading variable X (row 21), then the projection of the rectangle to y-axis is added to the new domain of Y (row 22). If it is the first rectangle that has a non-empty intersection with the current interval of X (row 23) then the start of the new interval is added to *ListOfX* (row 24). When entering the relevant rectangle we make this rectangle active by memorizing it in the *ActiveRects* structure (row 27).

If we leave a rectangle (*rect_end* event) that is the last active rectangle and the sweep line is within the domain of X (row 30) then the end of a new interval is added to *ListOfX* (row 31).

If we enter a new interval within the domain of X (*x_start* event) and there is any active rectangle (row 35) then the start of a new interval is added to *ListOfX* (row 36). Also, the new domain of Y is extended by the projections of the active rectangles to y-axis (rows 37-39).

If we leave an interval within the domain of X (*x_end* event) and there is still some active rectangle then the end of a new interval is added to *ListOfX* (row 43).

EVENT - ACTION

```
rect_start(PosX,NumR,IntY)
  20  if non_empty(intersection(IntY,DY)) then
  21     if InDomain then
  22      ListOfDomY ← IntY : ListOfDomY
  23      if empty(ActiveRects) then
  24         ListOfX ← PosX : ListOfX
  25      end if
  26     end if
  27     ActiveRects ← r(NumR,IntY) : ActiveRects
  28  end if
rect_end(PosXx,NumR)
  29  if find_and_delete(r(NumR,_),ActiveRects) then
  30     if InDomain && empty(ActiveRects) then
  31       ListOfX ← PosX : ListOfX
  32    end if
  33  end if
x_start(PosX)
  34  InDomain ← true
  35  if non_empty(ActiveRects) then
  36    ListOfX ← PosX : ListOfX
  37    for each r(NumR,IntY) in ActiveRects do
  38      ListOfDomY ← IntY : ListOfDomY
  39    end for
  40  end if
x_end(PosX)
  41  InDomain ← false
  42  if non_empty(ActiveRects) then
  43    ListOfX ← PosX : ListOfX
  44  end if
```

Fig. 13. The algorithms for event processing for the SP propagator.

Figure 14 gives an example of event processing. It describes how the data are changed after processing the events from the event point series.

Experiments and Comparison

We have compared the GR and SP propagators to existing `relation` and `case` constraints in SICStus Prolog 3.11.0 [9, 17]. The comparison was done using real-life scheduling problems solved by the Visopt ShopFloor system [2] and using a new artificial benchmark. The tests run under Windows XP Professional on 1.7 GHz Mobile Pentium-M 4 with 768 MB RAM and the running time is measured via the `statistics` predicate with the `walltime` parameter [17].

EVENT	***ListOfX***	***InDom.***	***ActiveRects***	***NewDY***
`rect_start(2,1,[2,5..6])`	`empty`	`false`	`r(1,[2.5..6])`	`empty`

`rect_start(3,2,[3..4])`	`empty`	`false`	`r(2,[3..4])`
			`r(1,[2,5..6])`
				`empty`

`X_start(3)`	`3`	`true`	`r(2,[3..4])`
			`r(1,[2,5..6])`
				`[2,5..6]`
				`[3..4]`

`rect_end(4,1)`	`3`	`true`	`r(2,[3..4])`	`[2,5..6]`
				`[3..4]`

`X_end(5)`	`5,3`	`false`	`r(2,[3..4])`	`[2,5..6]`
				`[3..4]`

`rect_start(7,3,[2,5..6])`	`5,3`	`false`	`r(3,[2,5..6])`
			`r(2,[3..4]`
				`[2,5..6]`
				`[3..4]`

`rect_end(7,2)`	`5,3`	`false`	`r(3,[2,5..6])`	`[2,5..6]`
				`[3..4]`

`X_start(8)`	`8,5,3`	`true`	`r(3,[2,5..6])`	`[2,5..6]`
				`[3..4]`

`rect_end(9,3)`	`9,8,5,3`	`true`	`empty`	`[2,5..6]`
				`[3..4]`

`X_end(10)`	`9,8,5,3`	`false`	`empty`	`[2,5..6]`
				`[3..4]`

Fig. 14. Example run of the SP filtering algorithm for the constraint domain from Figure 11. The constraint is not entailed and the domains are narrowed to DX=[3..5,8..9], DY=[2..6].

The GR and SP propagators are compared to `relation` and `case` constraints. The propagators behind these constraints maintain full arc consistency like the GR and SP propagators. The constraint domain in the `relation` constraint is described using a simple table $T = \{(x_i,dy_i) \mid i=1..n\}$, where x_i are pair-wise different values of the leading variable and dy_i is a range of values of the dependent variable compatible with the value x_i. Since the version 3.10.0 of SICStus Prolog, the `relation` constraint is implemented using a more general `case` constraint which, like our approach, allows more compact representation of the constraint domain. We use a table $CT = \{(dx_i,dy_i) \mid dx_i = \{x \mid (x,dy_i)\in T\} \ \&\ dx_i \neq \emptyset\}$ to describe the constraint domain for the `case` constraint – the actual representation contains one more element in the list representing CT, for details of syntax see [17]. This is exactly the same table used by the GR propagator. Note finally, that the `case` constraint is implemented in C while our propagators are implemented in Prolog.

Real-Life Experiments in Visopt ShopFloor

Visopt ShopFloor [2] is a scheduling system where the user describes declaratively the resources, item flows, and the demands and the system generates a schedule of production. Because of its generic character, the system uses many tabular constraints to describe the real-life relations like the time windows and the resource state transitions [1]. Originally, the GR propagator was designed for this system to capture a typical structure of constraint domains that appear there.

We have selected five different problems based on real-life factories to demonstrate capabilities of the proposed propagators. These problems vary in the size and the structure of the factories – the actual data are confidential so it is not possible to publish them. Table 1 describes the size of the problems as a number of different constraint domains (tables), a number of tabular constraints using these domains, and an average size of the constraint domain representation. The size of the representation is measured as an average number of rectangles per table for GR and SP propagators and as an average length of the lists describing the domains in the `relation` and `case` constraints.

Table 1. The size of the test problems and constraint representations.

problem no.	tables	constraints		average representation size per table			
		total	per table	GR	SP	relation	case
1	401	16977	42	1.13	4.40	20.76	2.13
2	49	1921	39	3.08	15.78	39.57	4.08
3	158	5734	36	2.32	20.27	44.82	3.32
4	244	82804	339	1.20	1.80	3.60	2.10
5	112	7624	68	1.04	1.59	3.65	2.04

Notice, that many constraints share the same domain, for example more than three hundred constraints share the domain (in average) in the problem no 4. This domain sharing decreases memory consumption. Moreover, the domain generator for both GR and SP propagators runs once per table which decreases running time. Notice also that the domain representation is very compact for GR and SP propagators. The only exception is the domain representation for the SP propagator in problems 2 and 3. A compact representation further reduces the running time because the time complexity of GR and SP propagators depends on the number of rectangles in the representation [3,4]. As expected, the representation for the GR propagator is more compact than the representation for the SP propagator.

Table 2 compares the number of calls to GR and SP propagators with and without the entailment detector (this information is not available for the built-in `relation` and `case` constraints). When the entailment detector is not used (off) then the number of calls is identical for both GR and SP because this number is influenced by the environment, where the algorithm sits, not by the algorithm itself. When the entailment detector is used (on) then the number of calls is much smaller. Notice also that for the problems 2, 3, and 4, the number of calls to SP is larger than the number of calls to GR. This indicates that the entailment detector for SP is not complete.

Table 2. The number of calls to the propagators. The number in brackets indicates the number of calls relative to the number of calls when the entailment detector is used (on).

problem no.	GR		SP	
	on	off	on	off
1	18515	117328 (634%)	18515	117328 (634%)
2	3372	9062 (269%)	3436	9062 (264%)
3	16688	56413 (338%)	19072	56413 (296%)
4	82830	145654 (176%)	86265	145654 (169%)
5	8014	32301 (403%)	8014	32301 (403%)

Finally, Table 3 compares the running times of the algorithms. The average time of five runs for each problem is indicated in the table. Note also, that we compare a total running time to solve the problem including propagation in all constraints as well as search. Thus, the actual running time of the compared algorithms is just a fraction of the presented time. Still, only the compared algorithms are responsible for the difference in the running time so the relative time difference between the algorithms is higher than it might seem from Table 3. We decided for this test because it shows better what speed-up/slow-down one may expect in a complex system.

Table 3. The running time (in seconds) of the propagators. The numbers in brackets show time relative to the GR propagator with entailment detector (in percent).

problem	GR		SP		relation	case
no.	on	off	on	off		
1	82,0	81,7 (100%)	81,5 (99%)	88,8 (108%)	91,5 (112%)	89,0 (108%)
2	3,6	3,7 (102%)	4,6 (127%)	5,0 (138%)	4,2 (117%)	4,1 (114%)
3	48,6	53,8 (111%)	57,8 (119%)	66,6 (137%)	66,6 (137%)	62,2 (128%)
4	86,1	87,8 (102%)	88,4 (103%)	90,4 (105%)	96,2 (112%)	94,6 (110%)
5	26,5	26,3 (99%)	26,4 (100%)	27,7 (105%)	35,0 (132%)	37,9 (143%)

The GR propagator with entailment detector achieved the best results in the tests. It has the smallest running time for the problems 2, 3, and 4 and its running time is very close to the best results in tests 1 (SP on) and 5 (GR off).

The behavior of the SP propagator with entailment detector was less stable. In particular, it achieved much worse running time than GR in tests 2 and 3. However, recall that the representation of the constraint domain is less compact for SP than for GR (Table 1) and better domain generator may further improve time efficiency of SP. Moreover, the SP propagator uses simpler elementary operations than GR which is the reason why SP is better in some tests despite the fact that it has a larger domain representation than GR.

The tests also show that entailment detection pays off especially for the SP propagator where entailment detection brings almost no overheads. The entailment detector for GR is more complex and it adds more overhead to computation. That is the reason why the GR propagator without entailment detection may get slightly better results in some problems.

Notice finally, that the built-in `relation` and `case` constraints implemented in C achieved worse results than GR and SP (with the exception of problem 2 for SP) implemented in Prolog. The reason could be that GR is designed for tabular constraints which have almost rectangular structure. However, note that we use the same compact representation for the `case` constraint.

The real-life tests justified the choice of the GR propagator with entailment detector in the Visopt ShopFloor system. However, they did not uncover the general features of the new propagators like the relation between the compaction factor and the running time. Therefore, we proposed an artificial benchmark to test the comparators independently of a particular application.

Artificial Benchmarks

Artificial benchmarks help to understand general features of the algorithms without a direct relation to a particular application. Random CSP [13] is a widely accepted set of benchmarks for testing constraint satisfaction algorithms. However, the disadvantage of Random CSP is that there is no direct relation between the parameters of the random problem, like density and tightness, and the structure of the constraint domain. Because time and space complexity of the propagators studied in this paper depends strongly on the structure of the constraint domain, we decided to design a new random benchmark where the number of rectangles in the constraint domain can be controlled by a parameter.

The basic idea of the proposed benchmark is to use a single binary constraint with a randomly generated domain. The size of domains for the variables is a parameter of the benchmark; we decided for the size 10 000 because we studied propagators for large domains. For each value of the leading variable, we randomly generate an interval of the compatible values for the dependent variable. The length of this interval is another parameter of the benchmark; we tested lengths from 1000 to 9000 with the step 1000. Note, that the length of the interval is identical for each value of the leading variable, only the position of the interval is generated randomly. Thus, the larger interval implies a smaller number of possible positions which further implies a smaller number of rectangles in the constraint domain. Figure 15 shows the relation between the length of the interval and the size of the domain representation – the number of rectangles. For each length we randomly generated ten constraint domains and we present the average results over these constraint domains.

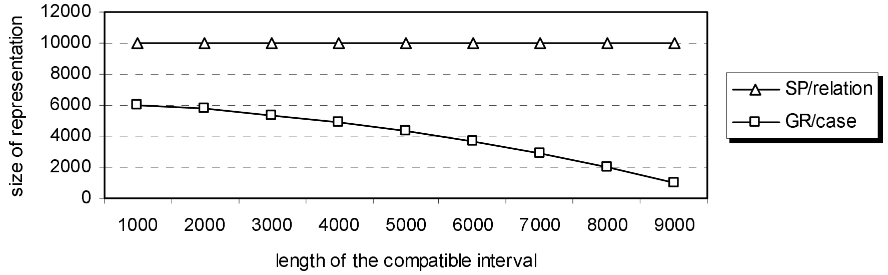


Fig. 15. A size of the domain representation for GR and SP propagators.

As we can see from Figure 15, the number of rectangles for the GR propagator is continuously decreasing with the increasing length of the interval. Unfortunately, the domain generator for the SP propagator does not compact the domain at all so we cannot expect good efficiency of the SP propagator in the following tests.

When the constraint domain is generated, the question is how to evoke the propagator. In the AC-3 (AC-8) schema, the propagator is evoked when the domain of any variable in the constraint is changed. So, we randomly prune the domains of the variables until one of these domains contains exactly one value. Then the constraint is entailed. The leading and dependent variables alternate in this process to suppress the role of the variable. By using this technique, we can measure the time spent only in the filtering algorithm. In particular, such benchmark abstracts from the complexity of the constraint satisfaction problem and the size of the search space is irrelevant here. Still, the question how to prune the variables' domains remains unanswered. We have tested three different schemas of domain pruning: domain splitting, arbitrary deletions, and shaving.

Domain Splitting. In domain splitting, the variable's domain is randomly split into two parts and one of these parts is pruned. In particular, we generate a random value called a cutting point between the lower and upper bound of the variable domain and we randomly decide whether to cut the lower or upper part of the domain with respect to the cutting point. Figure 16 shows the running time (in milliseconds) as a function of the length of the compatible interval.

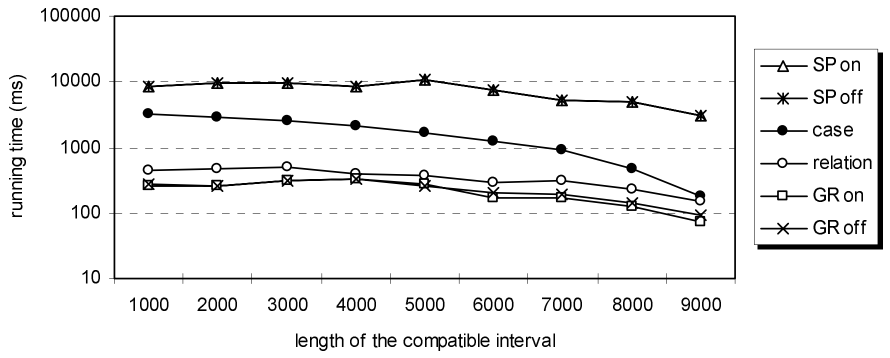


Fig. 16. The running time (a logarithmic scale in milliseconds) as a function of the length of the compatible interval for domain splitting.

For domain splitting, the GR propagator is the fastest propagator among the tested algorithms. It is about two times faster than the `relation` constraint (note, that we use a logarithmic scale in Figure 16). We can also see that entailment detection for GR pays off when the domain is more compacted. Surprisingly, the `case` constraint is not as good, even if it uses the same domain representation as the GR propagator. Nevertheless, the `case` constraint is becoming more efficient when the domain representation is more compact. The SP propagator does not behave well, probably because of the large domain representation.

Arbitrary Deletions. In many constraint satisfaction problems, the values are deleted from all over the domain. To capture this situation, we randomly generate a given number of values in the variable's domain and, then, we remove these values together from the domain. The number of values for deletion is given in percent of the current size of the domain. We tested four scenarios with 5%, 10%, 20%, and 40% of values deleted together from the domain. Figure 17 shows the running time (in milliseconds) as a function of the length of the compatible interval for all these scenarios.

For arbitrary deletions, the `relation` constraint is the best followed by the `case` constraint. Again, for more compacted representation, the `case` constraint is closer to the `relation` constraint. The GR propagator did not behave very well in these tests. The reason could be that the compact representation of the GR propagator works better when intervals of values are pruned rather than when individual values are deleted from domains. Notice also that the more values are removed together, the closer the GR propagator is to the `relation` and `case` constraints. Entailment detection has almost no effect here.

Shaving. In some applications, like scheduling, the domains of variables are pruned in a specific way. In particular, upper or lower parts of the domains are pruned which we call shaving. This technique is close to domain splitting but we can now control better the number of deleted values. In particular, we shave a given percent of the domain, namely 5%, 10%, 20%, and 40%. The choice whether to shave upper or lower part of the domain is done randomly. Figure 18 shows the running time (in milliseconds) as a function of the length of the compatible interval.

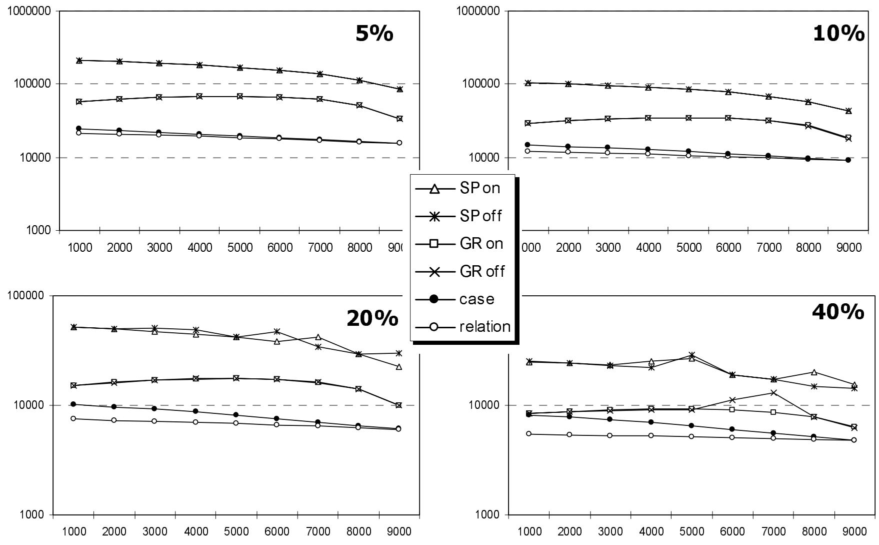


Fig. 17. The running time (a logarithmic scale in milliseconds) as a function of the length of the compatible interval for arbitrary deletions. A given percent of randomly selected values is deleted from the domain.

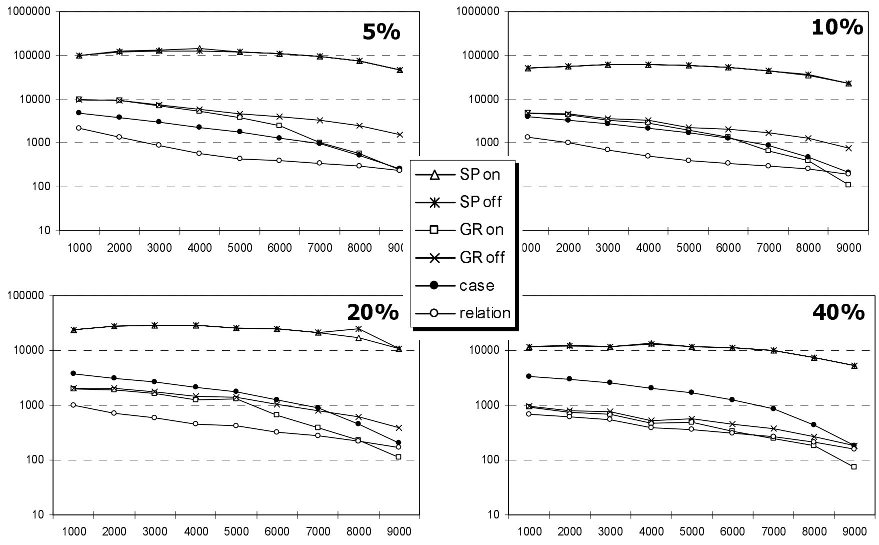


Fig. 18. The running time (a logarithmic scale in milliseconds) as a function of the length of the compatible interval for shaving. A given percent of values is shaved from the domain.

The power of the GR propagator is visible in the tests with shaving. Actually, the GR propagator outperforms the `relation` constraint when the constraint domain is more compacted and when more values are shaved. This trend is even stronger when we compare the GR propagator with the `case` constraint. Finally, notice that the entailment detector for GR also pays off significantly in these cases.

Conclusions

The paper proposed and compared two approaches to domain filtering for extensionally defined binary constraints, namely GR and SP propagators. Both approaches are based on the idea of compact representation of the constraint domain as a set of rectangles. They differ in the structure of these rectangles and in the way how this structure is explored during filtering. We presented the filtering algorithms as well as the algorithms for constructing a domain representation. We also described entailment detection mechanism and we showed that it improves real performance of the propagators. The experimental comparison showed that efficiency of the proposed propagators depends strongly on the size of domain representation. Thus, the future research may go in the direction of designing smaller decompositions of the constraint domain especially for the SP propagator. Also, both propagators explore the constraint domain completely after variable's domain change which penalizes them when just a few values are deleted. It could be interesting to reduce the number of compatibility checks during filtering, for example using information about the cause of calling the propagator like in [8, 20].

References

1. Barták, R.: Modelling Resource Transitions in Constraint-Based Scheduling. In Grosky W.I., Plášil F. (eds.): Proceedings of SOFSEM 2002: Theory and Practice of Informatics, LNCS 2540, Springer Verlag (2002), pp. 186-194.
2. Barták, R.: Visopt ShopFloor: On the edge of planning and scheduling. In Van Hentenryck P. (ed.): Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, LNCS 2470, Springer Verlag (2002), pp. 587-602.
3. Barták, R.: Filtering Algorithms for Tabular Constraints. In Proceedings of Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS), Paphos (2001), pp. 168-182.
4. Barták, R.: A General Relation Constraint: An Implementation. In Proceedings of CP2000 Post-Workshop on Techniques for Implementing Constraint Programming Systems (TRICS), Singapore (2000), pp. 30-40.
5. Beldiceanu N., Carlsson M.: Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Walsh T. (ed.): Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, LNCS 2239, Springer Verlag (2001), pp. 377-391.
6. Bessière Ch.: Arc-consistency and arc-consistency again. Artificial Intelligence 65 (1994), pp. 179-190.
7. Bessière Ch., Freuder E.C., and Régin J.-Ch.: Using constraint metaknowledge to reduce arc consistency computation. Artificial Intelligence 107 (1999), pp. 125-148.
8. Bessière Ch. and Régin J.-Ch.: Refining the Basic Constraint Propagation Algorithm. In Proceedings of JFPLC'2001 (2001).
9. Carlsson M., Ottosson G., Carlsson B.: An Open-Ended Finite Domain Constraint Solver. In Proceedings Programming Languages: Implementations, Logics, and Programs (1997).
10. Carlsson, M., and Schulte, Ch.: Finite-Domain Constraint Programming Systems. Tutorial at CP 2002.

11. Cheng K.C.K., Lee J.H.M., and Stuckey P.J.: Box constraint collections for adhoc constraints. In Rossi F. (ed.): Proceedings of the 9th International Conference on Principles and Practices of Constraint Programming, LNCS, Springer-Verlag (2003).
12. Chmeiss A., Jégou P.: Efficient Path-Consistency Propagation. International Journal of Artificial Intelligence Tools 2 (1998), pp. 121-142.
13. Gent I.P., MacIntyre E., Prosser P., Smith B.M., and Walsh T.: Random constraint satisfaction: Flaws and structure. Technical Report APES-08-1998, APES Research Group (1998).
14. Mackworth, A.K.: Consistency in Networks of Relations. Artificial Intelligence 8 (1977), pp. 99-118.
15. Michalský, R.: Algorithms for Constraint Satisfaction, Master Thesis, Charles University, Prague (2001).
16. Mohr R., Henderson T.C.: Arc and Path Consistency Revised. Artificial Intelligence 28 (1986), pp. 225-233.
17. SICStus Prolog 3.11.0 User's Manual, SICS (2003).
18. Shearer J.B, Wu, S.Y., and Sahni S.: Covering Rectilinear Polygons by Rectangles. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 9 (1990).
19. Van Hentenryck P., Deville Y., and Teng C.-M.: A generic arc-consistency algorithm and its specializations. Artificial Intelligence 57 (1992), pp. 291-321.
20. Zhang Y., Yap R.: Making AC-3 an Optimal Algorithm. In Proceedings of IJCAI-01 (2001), pp. 316-321.

Constraint Methods for Modal Satisfiability

Sebastian Brand[1], Rosella Gennari[2], and Maarten de Rijke[3]

[1] CWI, Amsterdam, The Netherlands
Sebastian.Brand@cwi.nl
[2] ITC-irst, Trento, Italy
gennari@itc.it
[3] Language and Inference Technology Group, ILLC, U. of Amsterdam, The Netherlands
mdr@science.uva.nl

Abstract. Modal and modal-like formalisms such as temporal or description logics go beyond propositional logic by introducing operators that allow for a guarded form of quantification over states or paths of transition systems. Thus, they are more expressive than propositional logic, yet computationally better behaved than first-order logic. We propose constraint-based methods to model and solve modal satisfiability problems. We model the satisfiability of basic modal formulas via appropriate sets of finite constraint satisfaction problems, and then resolve these via constraint solvers. The domains of the constraint satisfaction problems contain other values than just the Boolean 0 or 1; for these values, we create specialised constraints that help us steer the decision procedure and so keep the modal search tree as small as possible. We show experimentally that this constraint modelling gives us a better control over the decision procedure than existing SAT-based models.

1 Introduction

In many areas of artificial intelligence and computer science, trees, graphs, transition systems, and other types of relational structures provide the natural mathematical means to model evolving systems or to encode information about such systems. One may have to deal with such structures for a variety of reasons, e.g., to evaluate queries, to check requirements, or to make implicit information explicit. Modal and modal-like logics such as temporal logic and description logic [6] provide a convenient and computationally well-behaved formalism in which such reasoning may be represented [12].

Driven by the increased *computational* usage and usefulness of modal and modal-like logics, the past decade has seen a wide range of initiatives aimed at developing, refining, and optimising algorithms for solving the satisfiability problem of basic modal logic. This has resulted in a series of implementations. Some of these implement special purpose algorithms for modal logic, while others exploit existing tools or provers for either first-order logic or propositional logic through some encoding. In this paper we follow the second approach: we put forward a proposal to model and solve the modal satisfiability problem as a set of constraint satisfaction problems.

Specifically, we stratify a modal satisfiability problem, which is PSPACE-complete, into layers of "simpler" constraint satisfaction problems, which are NP-complete. On top of this, we add a refinement that exploits the restricted syntactic nature of modal problems, and that enables us to make efficient use of existing constraint solvers to decide modal satisfiability. Using the constraint logic programming system ECL^{i}PS^e [27], we inherit for free all the solvers for finite constraint satisfaction problems (e.g., generalised arc-consistency plus backtracking) and primitive constraints (e.g., at_most_one) already implemented in ECL^{i}PS^e; hence we can run them "as is" on top of our modelling to decide modal satisfiability. While we cannot yet fully compete with today's highly optimised modal provers, our experimental evaluations suggest that the approach is very promising in general, and even excellent in some cases.

The main contributions of our work derive from our modelling of modal satisfiability problems: modal formulas are translated into layers of finite constraint problems that have domains with possibly *further values than the Boolean* 0 or 1 (see Section 4), together with *appropriate constraints* to reason about these values (see Section 5). As amply shown and discussed in Sections 5 and 6 below, our modelling has a number of benefits over existing encodings of modal formulas into sets of propositions. For instance, the extended domains together with appropriate constraints give us a *better control over the modal search procedure*: they allow us to set strategies on the variables to split on in the constraint solver in a compact manner. In particular, by means of appropriate constraints and heuristics for our model, we can obtain partial Boolean assignments instead of total assignments; see Subsection 5.2.

The rest of the paper is organized as follows. After having provided background material concerning the motivation of the work reported here and work related to ours in Section 2, we lay the propositional groundwork in Section 3. We turn to modal matters in Section 4. The main contributions of this paper start in Section 5, which presents our constraint model. Then in Section 6, we report on an experimental assessment on a benchmark test set used in the TANCS '98 comparison of provers for modal logic. We conclude in Section 7.

2 Background

In this section we address two aspects of our work. First, we provide some motivations for studying the satisfiability problem for modal and modal-like logic. And second, we relate our approach to existing work in the literature.

Motivations. We have a broad view of what modal logic is. On this view, modal logic encompasses such formalisms as temporal logic, description logic, feature logic, dynamic logic... While originating from philosophy, for the past three decades the main innovations in the area of modal logic have come from computer science and artificial intelligence. The modern, computationally motivated view of modal logic is one that takes modal logics to be expressive, yet computationally well-behaved fragments of first-order or second-order logic. Other computer science influences on modal logic include the introduction of many new formalisms, new algorithms for deciding reasoning tasks, and, overall, a strong focus on the interplay between expressive power and computational complexity. We now give examples of modern computational uses of modal-like logics.

We start with a brief look at the use of modal-like logics in the area of formal specification and verification; see [18] for a comprehensive introduction. Requirements such as "the system is `always` dead-lock free" or "the system `eventually` waits for a signal" can be compactly expressed in the basic modal logic by augmenting propositional logic with two operators: $\Box$ for the guarded universal quantifier over states (commonly read as `always`, meaning "in all the reachable states"), and $\Diamond$ for its existential counterpart (commonly read as `eventually`, meaning "in some reachable state"). Formalising "the system is dead-lock free" with the proposition *s_free* and "the system waits for a signal" with *s_wait*, the above two requirements correspond to the modal formulas $\Box s\_free$ and $\Diamond s\_wait$, respectively.

So-called multi-modal logics are popular in the agent-based community (e.g., see [24]); here, each agent is endowed with beliefs and knowledge, and with goals that it needs to meet. The beliefs and knowledge can be expressed by means of multi-modal operators: $\Box^b_A$ for "agent A believes" and $\Diamond^b_B$ for "agent B disbelieves"; $\Box^k_B$ for "agent B knows" and $\Diamond^k_A$ for "agent A ignores". More complex modal formulas involving until operators or path quantifiers are used to reason about agents' plans, in particular to express and verify specifications on plans (see, e.g., [5]) or extended goals (see, e.g., [23]).

Description logics are a family of modal-like logics that are used to represent knowledge in a highly structured manner [3], using (mostly) unary and binary relations on a domain of objects. Knowledge is organized in terminological information (capturing definitions and structural aspects of the relations) and assertional information (capturing facts about objects in the domain being modelled). For instance, an object satisfies $\Diamond_R A$ if it is R-related to some object satisfying A. In the area of description logic, a wide range of algorithms has been developed for a wide variety of reasoning tasks.

While there are many more areas where modal-like logics are currently being used, including semi-structured data [19], game theory [13], or mobile systems [7], due to space limitations we have to omit further details. What all of these computational applications of modal-like logics have in common is that they use relational structures of one kind or another to model a problem or domain of interest, and that a modal-like logic is used to reason about these structures. Moreover, for many of the above applications, *modal satisfiability checking* is the appropriate reasoning task: given a modal formula, is there a model on which this formula can be satisfied? In this paper we propose a new, constraint-based method for checking modal satisfiability.

Related Work. The past decade has seen a wide range of initiatives aimed at developing, refining, and optimising algorithms for solving the satisfiability problem of basic modal logic. Some of these implement special purpose algorithms for modal logic, such as DLP [22], FaCT [15], RACER [11], *SAT [25], while others exploit existing tools or provers for either first-order logic (MSPASS [20]) or propositional logic (KSAT [10], KBDD [21]) through some encoding. In this paper we follow the second approach: we propose to model and solve modal satisfiability problems as constraint problems.

The starting-points of our work are [10] and [2]. In [10], modal formulas are modelled and solved as sets of propositions (i.e., Boolean formulas) stratified into layers; the propositions are processed starting from the top layer in a depth-first left-most manner.

But we add a refinement that builds on ideas due to [2]. In [2] a refinement of an existing encoding of modal formulas into first-order formulas was introduced. This refinement enables one to re-use existing first-order theorem provers for deciding modal satisfiability, and, at the same time, to 'inform' the prover about the restricted syntactic nature of first-order translations of modal formulas, which resulted in a significant improvement in performance. We build on this intuition: we improve on the modelling of modal formulas in [10] so as to be able to make efficient use of existing constraint solvers to decide modal satisfiability. Specifically, modal formulas are translated into layers of finite constraint satisfaction problems that have domains with possibly further values than the Boolean 0 or 1, together with appropriate constraints to reason about these values. The well-known DPLL algorithm can also return partial Boolean assignments for propositions. But, in this respect, there are two key add-ons of our modelling. First, the use of extended domains and constraints allow us more control over the partial assignments to be returned by the adopted constraint solver than unit propagation allows for in DPLL. And second, we can run any constraint solver on top of our modelling to obtain partial assignments, i.e., it is by modelling that we obtain partial assignments, and not by modifying existing constraint solvers nor by choosing a specific solver to do so such as DPLL.

3 Propositions as Finite Constraint Problems

Constraint Satisfaction Problems. We begin with constraint satisfaction terminology. Consider a set $X := \{x_1, \ldots, x_n\}$ of n variables, and assume that X is ordered by $\prec$; a *scheme* of X is a sequence $s : x_1, \ldots, x_m$ of variables in X, where $x_{j-1} \prec x_j$ for each $j = 2, \ldots, m$. Associate one set D_i with each variable $x_i \in X$; then D_i is the *domain of* x_i; let $\boldsymbol{D}$ be the set of all such domain and variable pairs $\langle D_i, x_i \rangle$. Given a scheme $s := x_1, \ldots, x_m$, a relation $C(s)$ on the Cartesian product $\prod_{j=1}^{m} D_j$ is a *constraint on* s; let $\boldsymbol{C}$ be a set of constraint and scheme pairs $\langle C(s), s \rangle$ on X. Then $\langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ is a *constraint satisfaction problem* (CSP). A CSP is *finite* if all D_i in $\boldsymbol{D}$ are so. A tuple $d \in D_1 \times \cdots \times D_n$ is *consistent* or *satisfies* a constraint $C(s)$ if the projection of d on s, denoted by $\Pi_s(d)$, is in $C(s)$; if d satisfies all the constraints of the CSP P, then P is a *consistent* or *satisfiable CSP*. The projection of a constraint $C(t)$ over a subscheme s of t is denoted by $\Pi_s(C(t))$. Finally, a *total assignment for a CSP* $\langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ is a function $\mu : X \mapsto \bigcup_{i=1}^{n} D_i$ that maps each $x_i \in X$ to a value in the domain D_i of x_i; μ *satisfies the CSP* if the tuple $(\mu(x_1), \ldots, \mu(x_n))$ does so.

Propositions. When a Boolean-valued assignment μ satisfies a propositional formula ϕ, we write $\mu \models \phi$. We write $\mathit{CNF}(\phi)$ for the result of ordering the propositional variables in ϕ and transforming ϕ into a conjunctive normal form: i.e., a conjunction of disjunctions of literals without repeated occurrences; a *clause* of ψ is a conjunct of $\mathit{CNF}(\psi)$.

From Propositions to CSPs. It is not difficult to transform a propositional formula into a CSP so that this is satisfiable iff the formula is: first the formula is transformed to its CNF; then each resulting clause is considered as a constraint. E.g., the CNF formula

$$(\neg x \vee y \vee z) \wedge (x \vee \neg y) \tag{1}$$

is the CSP with variables x, y and z, domains equal to $\{0, 1\}$, and two constraints: $C(x, y, z)$ for $\neg x \vee y \vee z$, that forbids the assignment $\{x \mapsto 1, y \mapsto 0, z \mapsto 0\}$; and the constraint $C(x, y)$ for $x \vee \neg y$ to rule out the assignment $\{x \mapsto 0, y \mapsto 1\}$. In [28] the encoding in (1) is used to prove that a version of forward checking performs more inferences than the basic `DP` procedure for deciding propositional satisfiability.

However, a constraint solver returns a total assignment given the modelling of formulas as CSPs above, while we aim at *partial* Boolean assignments. For example, a partial assignment of only two variables suffices to satisfy the formula in (1), such as $\{x \mapsto 1, z \mapsto 1\}$. How do we get such without modifying the underlying constraint solver? One way is to encode the propositional formula into a CSP with values other than 0 and 1. The additional values are then used to mark variables that the solver does not need to satisfy (yet). Let us give a precise definition of this new encoding. We assume an implicit total order on the variables in the considered propositions; so, we identify formulas that only differ in the order of occurrence of their atoms, such as $y \vee x$ and $x \vee y$.

Definition 1. Given a propositional formula ψ, $\mathit{CSP}(\psi)$ is the CSP associated with ψ defined as follows:

1. construct $\psi' := \mathit{CNF}(\psi)$ and let X be the ordered set of propositional variables occurring in ψ';
2. create a domain $D_i := \{0, 1, \mathsf{u}\}$ for each x_i in X;
3. for each clause θ in ψ', there is a constraint C_θ on the scheme $s := x_1, \ldots, x_m$ of all the variables of θ; a tuple $d := (d_1, \ldots, d_m)$ in $\prod_{j=1}^{m} D_j$ satisfies C_θ iff there is a non-empty subscheme $s' := x_{i_1}, \ldots, x_{i_n}$ of s such that $d_{i_k} \in \{0, 1\}$ for all $k = 1 \ldots n$ and $d' := \Pi_{s'}(d)$ satisfies $\Pi_{s'}(C_\theta)$.

In Definition 1, we do not give any details on how constraints are represented and implemented; this is done on purpose, since these are not necessary for our theoretical results concerning the modal satisfiability solver. Nevertheless, some modelling choices and implementation details are discussed in Section 5 below.

Our modelling of propositional formulas as in Definition 1 allows us to make *any complete solver for finite CSPs return a partial Boolean assignment* that satisfies a propositional formula ψ iff ψ is satisfiable. To prove this, we need some notational shorthands. Given $CSP(\psi)$ on X as in Definition 1 above, let μ be a total assignment for $CSP(\psi)$, and $X|_{Bool}$ the subset of all $x_i \in X$ for which $\mu(x_i) \in \{0, 1\}$. Then the restriction of μ to $X|_{Bool}$ is denoted by $\mu|_{Bool}$: i.e., $\mu|_{Bool} : X|_{Bool} \mapsto \{0, 1\}$ and $\mu|_{Bool}(x_i) = \mu(x_i)$ for each $x_i \in X|_{Bool}$.

Theorem 1. *Consider a propositional formula ψ and let X be its ordered set of variables.*

1. *a total assignment μ for $CSP(\psi)$ satisfies $CSP(\psi)$ iff $\mu|_{Bool}$ satisfies ψ;*
2. *ψ is satisfiable iff a complete constraint solver for finite CSPs returns a total assignment μ for $CSP(\psi)$ such that $\mu|_{Bool}$ satisfies ψ.*

Proof. First notice that a proposition and its CNF are equivalent; in particular, a Boolean assignment satisfies the one iff it satisfies the other. Then item 1 follows from this, Definition 1 and a property of CNF formulas: a partial Boolean assignment μ satisfies $CNF(\psi)$ iff, for each clause ϕ of $CNF(\psi)$, μ assigns 1 to at least one positive literal in ϕ, or 0 to at least one negative literal in ϕ. Item 2 follows from the former. ●

Note 1. It is sufficient that each domain D_i of $CSP(\psi)$ contains the Boolean values 0 and 1 for the above result to hold. Thus, one could have values other than u (and 0 and 1) in the CSP modelling to mark some variables with different "levels of importance" for deciding the satisfiability of a propositional formula. However, our choice as in Definition 1 will suffice for the purposes in this paper.

4 Modal Formulas as Layers of Constraint Problems

In this section we recall the basics of modal logic and provide a link between solving modal satisfiability and CSPs.

4.1 Modal Formulas as Layers of Propositions

We refer to [6] for extensive details on modal logic. To simplify matters, we will focus on the basic mono-modal logic $\mathcal{K}$, even though our results can easily be generalized to a multi-modal version.

Modal Formulas. $\mathcal{K}$-formulas are defined as follows. Let P be a finite set of propositional variables. Then $\mathcal{K}$-formulas over P are produced by the rule

$$\phi \quad ::= \quad p \mid \neg\phi \mid \phi \wedge \phi \mid \Box\phi$$

where $p \in P$. The formula $\Diamond p$ abbreviates $\neg\Box\neg p$, and the other Boolean connectives are explained in terms of $\neg, \wedge$ as usual. For instance, all of p, q, $p \vee q$, $\Box(p \vee \Box q) \wedge \Diamond\Box p$ are $\mathcal{K}$-formulas over $\{p, q\}$. A formula of the form $\Box\phi$ is called a *box formula.*

Note 2. Here and in the remainder, we always assume that P is implicitly ordered to avoid modal formulas only differing in the order of their propositional variables; also, standard propositional simplifications such as the removal of double occurrences of $\neg$ are implicitly performed on modal formulas.

Modal Layers and Propositional Approximations. The satisfiability procedure for $\mathcal{K}$-formulas in this paper (see Subsection 4.2 below) revolves around two main ideas:

- the stratification of a modal formula into layers of formulas of decreasing "modal depth";
- the "approximation" and resolution of such formulas as propositions.

Let us make those ideas more precise, starting with the former. The "modal depth" of a formula counts its maximum number of nested boxes, that is it measures "how deeply" we can descend into the formula by peeling off nested boxes. Formally, the *modal depth* of ϕ, denoted by $\mathrm{md}(\phi)$, is defined as follows:

$$\begin{aligned} \mathrm{md}(p) &:= 0 & \mathrm{md}(\Box\phi) &:= \mathrm{md}(\phi) + 1 \\ \mathrm{md}(\neg\phi) &:= \mathrm{md}(\phi, i) & \mathrm{md}(\phi_1 \wedge \phi_2) &:= \max\{\mathrm{md}(\phi_1), \mathrm{md}(\phi_2)\}. \end{aligned}$$

For instance, consider $\phi = \Box p \vee \neg q$ which intuitively means that "`always` p or q fails." Then $\mathrm{md}(q, \phi) = 0$, $\mathrm{md}(\Box p) = \mathrm{md}(p) + 1 = 1$, hence $\mathrm{md}(\phi) = 1$; in other words, the maximum number of nested boxes we can peel off from ϕ is 1. Testing if a modal formula is satisfiable involves stratifying it into layers of subformulas (or Boolean combinations of these) of decreasing modal depth. At each such layer, modal formulas get "approximated" and solved as propositions. Formally, given a modal formula ϕ, the *propositional approximation* of ϕ, denoted by $\mathit{Prop}(\phi)$, is the proposition inductively defined as follows:

$$\begin{aligned} \mathit{Prop}(p) &:= p & \mathit{Prop}(\Box\phi) &:= x_i[\Box\phi] \\ \mathit{Prop}(\neg\phi) &:= \neg\mathit{Prop}(\phi) & \mathit{Prop}(\phi_1 \wedge \phi_2) &:= \mathit{Prop}(\phi_1) \wedge \mathit{Prop}(\phi_2). \end{aligned}$$

We denote here by $x_i[\Box\phi]$ a fresh propositional variable that is associated with one occurrence of $\Box\phi$. Note that in this way different occurrences of $\Box\phi$ are distinguished by introducing different new variables. For instance: the modal formula $\phi = p \wedge \Box q \vee \neg\Box q$ is approximated by the proposition $\mathit{Prop}(\phi) = p \wedge x_1[\Box q] \vee \neg x_2[\Box q]$. The variables of ϕ are $\{p, q, r\}$, while $\mathit{Prop}(\phi)$ is over the variables $\{p, x_1[\Box q], x_2[\Box q]\}$.

Now that the ideas of modal depth and approximation of modal formulas as propositions are made precise, we can put them to work in the $\mathcal{K}$-satisfiability procedure below.

4.2 $\mathcal{K}$-Satisfiability and the General *k_sat* Schema

In Figure 1 below, we formalise $\mathcal{K}$-satisfiability and present the general algorithm schema *k_sat*, on which `KSAT` [10] is based, for deciding the satisfiability of $\mathcal{K}$-formulas.

```
k_sat($\phi$)       // succeeds if $\phi$ is satisfiable

        $\mu := \mathbf{sat}(Prop(\psi))$                  // create a choice point
        $\Theta := \bigwedge \{\theta \ : \ \text{if} \ \ x[\Box\theta] \mapsto 1 \ \text{ is in } \mu\}$
        for each  $x[\Box\lambda] \mapsto 0$  in $\mu$ do
                $k\_sat(\Theta \wedge \neg\lambda)$              // backtrack if this fails

sat($\phi$)         // if $\phi$ is propositionally satisfiable, then return a Boolean
                    // assignment; return alternatives on backtracking
```

Fig. 1. The *k_sat* algorithm schema.

$\mathcal{K}$-Satisfiability. At this point we have to make a choice between a more "standard" characterisation of the semantics of $\mathcal{K}$-formulas or another that is closer to the semantics of the solving algorithm. Here we choose for the latter which allows us to arrive more quickly and concisely at the matters of this paper.

Definition 2. The $\mathcal{K}$-formula ϕ is *$\mathcal{K}$-satisfiable* iff there is a Boolean assignment μ that satisfies $Prop(\phi)$, and for every $\Box\lambda$ with $\mu(x[\Box\lambda]) = 0$, the $\mathcal{K}$-formula

$$\neg\lambda \wedge \bigwedge \{\theta \ : \ \mu(x[\Box\theta]) = 1\}$$

is $\mathcal{K}$-satisfiable.

Algorithm Schema. In the *k_sat* schema, the *sat* procedure determines the satisfiability of the propositional approximation of ϕ by returning a Boolean assignment μ as in Definition 2. Alternative satisfying assignments are generated upon backtracking. If there is no alternative assignment, then this call to *k_sat* fails and backtracking takes place, except on the input formula where "formula is unsatisfiable" is reported. In this manner, the modal search space gets stratified into modal formulas of decreasing modal depth and explored in a depth-first manner. A variable of the form $x[\Box\lambda]$ to which μ assigns 0 means that we *must* "remove the box" and check λ against all the formulas θ that come with variables of the form $x[\Box\theta]$ to which μ assigns 1; precisely one proposition is so created and tested satisfiable.

Theorem 2. *In the k_sat algorithm schema in Figure 1, if sat is a complete solver for Boolean formulas, then k_sat is a decision procedure for $\mathcal{K}$-satisfiability.*

Proof. The characterisation of $\mathcal{K}$-satisfiability in Definition 2 is responsible for the correctness and completeness of *k_sat*; this terminates since the modal depth and the number of propositional variables of a modal formula are bounded. •

4.3 The KCSP Algorithm

We now devise a modal decision procedure based on the *k_sat* schema, but with a constraint solver as the underlying propositional solver *sat*. We first provide the reader with an example, and then formalise the procedure.

Example 1. Consider the following modal formula

$$\phi \;=\; \neg\Box(p \vee \bot) \;\wedge\; (\Box r \vee \Box p),$$

which intuitively means "it is never that p fails or that false holds, and it is `always` r or it is `always` p". Approximating ϕ as the proposition $\mathit{Prop}(\phi)$, the following CSP is obtained:

(a) three variables: $x[\Box(p \vee \bot)]$; $x[\Box r]$; $x[\Box p]$;
(b) three domains, all equal to $\{0, 1, \mathsf{u}\}$;
(c) two constraints: the one for $\Box(p \vee \bot)$ that forces the assignment 0 to $x[\Box p \vee \bot]$; the other for $(\Box r \vee \Box p)$ that requires 1 to be assigned to $x[\Box r]$ or $x[\Box p]$.

Assigning the value u to a variable means not committing to any decision concerning its Boolean values, 0 and 1. The above CSP is given to the constraint solver, and this may return the assignment

$$\mu = \{x[\Box(p \vee \bot)] \mapsto 0, x[\Box r] \mapsto \mathsf{u}, x[\Box p] \mapsto 1\}.$$

Then, for all the variables $x[\Box \ldots]$ to which μ assigns 1 (in this case only $x[\Box p]$), the formulas within the scope of $\Box$ are joined in a conjunction Φ, in this case

$$\Phi := p. \tag{UT}$$

Then all the box variables to which μ assigns 0 are considered, in this case only $x[\Box(p \vee \bot)]$; thus $p \vee \bot$ gets negated, simplified (translated in CNF when needed) and the result is the formula

$$\Theta := \neg p. \tag{ET}$$

The conjunction $\Phi \wedge \Theta$ is given to the *sat* solver; in this case, the clause that is passed on is $p \wedge \neg p$. This is translated into a new CSP and its inconsistency is determined. On subsequent backtracking, we may obtain μ' instead of the above assignment μ:

$$\mu' = \{x[\Box(p \vee \bot)] \mapsto 0, x[\Box r] \mapsto 1, x[\Box p] \mapsto \mathsf{u}\}.$$

The new (UT) $\Phi := r$ is created; the satisfiability of $\neg p \wedge r$ is determined, and thus that of ϕ. •

Notice the key points about (UT) and (ET): we *only* consider the box variables $x[\Box \ldots]$ to which a Boolean value, 0 or 1, is assigned. The box variables to which u is assigned are disregarded, safely so because of Theorem 1. As we will see below (Section 5), the availability of values other than 0 and 1 has a number of advantages.

As the above example illustrates, we first approximate ϕ as a proposition and then translate this into a CSP. Recall from Definition 1 how the CSP of a proposition ϕ is obtained: $\mathit{CSP}(\phi)$ is the CSP with domains containing another value than the Boolean 0 and 1.

Definition 3. The KCSP *algorithm* is defined as follows. In the *k_sat* schema we instantiate the *sat* function with a complete solver for finite CSPs and we preprocess ϕ into $CSP(Prop(\phi))$ before passing it on to the constraint solver.

Theorems 1 and 2 yield the following result concerning KCSP as in Definition 3.

Corollary 1. KCSP *is a decision procedure for $\mathcal{K}$-satisfiability.* •

In particular, notice again that modelling $Prop(\phi)$ as a CSP with an additional non-Boolean value allows us to instantiate *sat* to *any* constraint solver in KCSP and still obtain *partial* Boolean assignments.

5 Constraint-Based Modelling

In this section we discuss the constraints into which we translate a modal formula. We begin with a base modelling, and proceed to an improved modelling that possesses some desirable properties.

5.1 A Base Modelling

The input to KCSP is a formula in conjunctive normal form (CNF). Hence we translate a formula into a CSP clause-wise, each clause contributing one constraint (see Definition 3 above).

Aspect 1: Clauses as Constraints. For modelling a clause as a constraint, we distinguish four disjoint sets of variables: propositional variables and variables representing box formulas, and both subdivided according to polarity. We denote these sets P^+, P^-, B^+, B^-, respectively. This means a clause can be written as

$$\begin{array}{llll} \bigvee\{\, p : p \in P^+\} & \vee & \bigvee\{\, \neg p : p \in P^-\} & \vee \\ \bigvee\{\, x[\Box\phi] : x[\Box\phi] \in B^+\} & \vee & \bigvee\{\, \neg x[\Box\phi] : x[\Box\phi] \in B^-\} & \end{array}$$

This clause is viewed as a constraint on variables in the four sets:

$$clause\_constraint(P^+, P^-, B^+, B^-).$$

It holds if *at least one* variable in the set $P^+ \cup B^+$ is assigned a 1 or one in $P^- \cup B^-$ is assigned a 0 — see Definition 1 above. We explain now this constraint in terms of the primitive constraint at_least_one, which is defined on a set of variables and parametrised by a constant, and which requires the latter to occur in the variable set. This, or a closely related constraint, is available in many constraint programming languages. The constraint library of ECL^{i}PS^e [27] contains a predefined constraint with the meaning of at_most_one, which can be employed to imitate at_least_one. We reformulate the *clause_constraint* as the disjunction

$$\mathsf{at\_least\_one}(P^+ \cup B^+, 1) \quad \vee \quad \mathsf{at\_least\_one}(P^- \cup B^-, 0).$$

Aspect 2: Disjunctions as Conjunctions. Propagating disjunctive constraints is generally difficult for constraint solvers. Therefore it is preferable to avoid them when modelling; and in our situation we can do so elegantly. The disjunction is transformed into a conjunction with the help of a single auxiliary link variable $\ell \in \{0, 1\}$. We obtain

$$\mathsf{at\_least\_one}(P^+ \cup B^+ \cup \{\ell\}, 1) \quad \wedge \quad \mathsf{at\_least\_one}(P^- \cup B^- \cup \{\ell\}, 0).$$

The link variable ℓ selects implicitly which of the two constraints must hold. For example, observe that $\ell = 0$ selects the constraint on the left. It forces $\mathsf{at\_least\_one}(P^+ \cup B^+, 1)$ and satisfies $\mathsf{at\_least\_one}(P^- \cup B^- \cup \{0\}, 0)$. It is useful to remark the following fact.

Fact 1 *A conjunctive constraint built from two conjuncts that share at most one variable is generalised arc-consistent if its two constituent constraints are.* •

In our case the two conjuncts share no variables except ℓ.

5.2 A More Advanced Modelling

In our advanced modelling we add on additional features to the base modelling; those features are meant to address a number of aspects of the base modelling.

Aspect 3: Partial Assignments by Constraints. While any solution of the CSP induced by a formula at some layer satisfies the formula at that layer, it is useful to obtain satisfying, partial Boolean assignments that mark as irrelevant as many box formulas in this layer as possible. This will cause fewer subformulas to enter the propositions generated for the subsequent layer. In our model, we employ the extra value u to mark irrelevance. In such a CSP, consider a clause constraint c on propositional variables $P = P^+ \cup P^-$ and variables representing box formulas $B = B^+ \cup B^-$. By definition, the variables in B are constrained only by c. Given this and Theorem 1, we can conclude the following:

Fact 2 *Suppose μ is a partial assignment that can be extended to a total assignment satisfying the CSP. Suppose μ is not on the variables $P \cup B$.*

- *The assignment $\mu \cup \{p \mapsto 1\} \cup \{x[\Box\phi] \mapsto \mathsf{u} : x[\Box\phi] \in B\}$, where $p \in P^+$, satisfies c and can be extended to a total assignment satisfying the CSP. An analogous result holds for $p \in P^-$.*
- *The assignment $\mu \cup \{x[\Box\psi] \mapsto 1\} \cup \{x[\Box\phi] \mapsto \mathsf{u} : x[\Box\phi] \in B - \{x[\Box\psi]\}\}$, where $x[\Box\psi] \in B^+$, satisfies c and can be extended to a total assignment satisfying the CSP. Again, an analogous result holds for $x[\Box\psi] \in B^-$.* •

In other words, if satisfying the propositional part of a clause suffices to satisfy the whole clause, then all box formulas in it can be marked irrelevant. Otherwise, all box formulas except one can be marked irrelevant.

Let us transfer this idea into a clause constraint model. First, we rewrite the base model so as to

– separate the groups of variables (in propositional and box variables),
– and convert the resulting disjunctions into conjunctions, again with the help of extra linking variables.

Next, we replace the at_least_one constraint for variables representing box formulas by an exactly_one constraint. This simple constraint is commonly available as well. ECL^{i}PS^e offers the more general occurrences constraint, which forces a certain number of variables in a set to be assigned to a specific value. We obtain

$$\begin{array}{lclc} \mathsf{at\_least\_one}(P^+ \cup \{\ell_P^+\}, 1) & \wedge & \mathsf{exactly\_one}(B^+ \cup \{\ell_B^+\}, 1) & \wedge \\ \mathsf{at\_least\_one}(P^- \cup \{\ell_P^-\}, 0) & \wedge & \mathsf{exactly\_one}(B^- \cup \{\ell_B^-\}, 0). & \end{array}$$

The variable domains are: $P^+, P^- \in \{0,1\}$, $B^+ \in \{1, \mathsf{u}\}$, $B^- \in \{0, \mathsf{u}\}$. The essential four linking variables are constrained as in the following formula, or the equivalent table.

$$(\ \ell_P^+ = 1 \wedge \ell_P^- = 0\ ) \ \leftrightarrow\ (\ \ell_B^- = \mathsf{u}\ \vee\ \ell_B^+ = \mathsf{u}\ )$$
$$\wedge$$
$$\ell_B^+ = 1\ \vee\ \ell_B^- = 0$$

ℓ_P^+	ℓ_P^-	ℓ_B^+	ℓ_B^-
1	0	1	u
1	0	u	0
0	1	1	0
0	0	1	0
1	1	1	0

Observe that the 5 tuples in the table correspond to the situations that we wish to permit — the clause is satisfied by either a positive or a negative box formula (but not both at the same time) or a positive or a negative propositional variable (maybe both at the same time).

ECL^{i}PS^e accepts the linking constraint in propositional form, and rewrites it internally into several arithmetic constraints. Alternative methods operate on the defining table. For our implementation we compiled it into a set of domain reduction rules, which can be executed efficiently [1]. This proved to be the fastest way of propagating this constraint among several methods we tested. We found that this linking constraint, among all constraints, is the one whose propagation is executed most often, hence propagating it efficiently is relevant.

Aspect 4: A Negated-CNF Constraint. Except for the initial input formula to KCSP which is in conjunctive normal form, the input to an intermediate call to the *sat* function of KCSP (see the algorithm in Figure 1) has the form $\Theta \wedge \neg\lambda$ where both Θ and λ are in CNF. A naive transformation of $\neg\lambda$ into CNF will result in an exponential increase in the size of the formula. We deal with this problem by treating $\neg\lambda$ as a constraint. Then the following holds.

Fact 3 *The constraint $\neg\lambda$ is satisfiable iff λ (which is a conjunction of clauses) has at least one unsatisfiable clause.* •

We formulate the constraint corresponding to $\neg\lambda$ consequently as a disjunction of constraints, each standing for a negated clause. The disjunction is converted into a conjunction with a set L of linking variables, one for each disjunct. The

$\ell_i \in L$ have the domain $\{0, 1\}$, where $\ell_i = 1$ means that the i-th disjunct holds — that is, the i-th clause in λ is unsatisfied. Instead of imposing $\mathsf{at\_least\_one}(L, 1)$ to select one disjunct, however, we require $\mathsf{exactly\_one}(L, 1)$, in line with our goal of obtaining small partial Boolean assignments. In irrelevant disjuncts/clauses ($\ell_i = 0$) we force the box formulas to u. The definition for the constraint corresponding to a negated clause on sets of variables P^+, P^-, B^+, B^-, together with the linking variable ℓ, is

$$\begin{array}{lll} \ell = 1 & \leftrightarrow & (\,\forall b \in B^+.\ b = 0 \quad \wedge \quad \forall b \in B^-.\ b = 1\,) \\ \wedge & & \\ \ell = 1 & \rightarrow & (\,\forall p \in P^+.\ p = 0 \quad \wedge \quad \forall p \in P^-.\ p = 1\,) \\ \wedge & & \\ \ell = 0 & \leftrightarrow & (\,\forall b \in B^+.\ b = \mathsf{u} \quad \wedge \quad \forall b \in B^-.\ b = \mathsf{u}\,). \end{array}$$

There is no need to constrain the propositional variables in a clause that is not selected by $\ell = 1$. Therefore, the propositional variables in P^+, P^- occur here less often than the variables in B^+, B^- which represent box formulas.

In `KCSP`, the propagation of this constraint is implemented as a user-defined constraint in the constraint library of ECL^{i}PS^e, and achieves generalised arc-consistency.

Aspect 5: A Constraint for Factoring. In our base model, we have treated and constrained each occurrence of a box formula as a distinct propositional variable. For instance, the two occurrences of $\Box p$ in the formula $\Box p \wedge \neg\Box p$ would be treated as two distinct propositional variables in our base model. We consider here the case that a box formula occurs several times, in several clauses, in any polarity. We then prevent assigning conflicting values to different occurrences.

Let us collect in $B_{\Box\phi}$ all variables $x_i[\Box\phi]$ representing the formula $\Box\phi$ in the entire CSP. We state as a constraint on these variables that

$$\forall x_1, x_2 \in B_{\Box\phi}.\quad \neg\,(\quad x_1 = 1 \wedge x_2 = 0 \quad \vee \quad x_1 = 0 \wedge x_2 = 1 \quad).$$

To see the effect, suppose there is a pair $x_1, x_2 \in B_{\Box\phi}$ with $x_1 \mapsto 0, x_2 \mapsto 1$ in a solution to the CSP without this factoring constraint. This means we obtain both $\Box\phi \mapsto 0$ and $\Box\phi \mapsto 1$ in the assignment returned, which in turn results in an unsatisfiable proposition being generated. The factoring constraint detects such failures earlier. Notice that the straightforward modelling idea, namely using one unique variable for representing a box formula in all clauses, clashes with the assumption made for the other partial-assignment constraints, i.e. that each box formula variable is unique.

6 Experimental Assessment

Theoretical studies often do not provide sufficient information about the effectiveness and behaviour of complex systems such as satisfiability solvers and their optimisations. Empirical evaluations must then be used. In this section we provide an experimental comparison of our advanced modelling (Section 5.2) against the base model (Section 5.1), using a test developed by Heuerding and Schwendimann [14].

We will find that, no matter what other models and search strategies we commit to, we always get the best results by using constraints for partial assignments as in Subsection 5.2, Aspect 3. In the remainder of this paper, these are referred to as the *assignment-minimising constraints* or simply as the *minimising constraints*. As we will see below, these minimising constraints allows us to better direct the modal search procedure.

We conclude this section by comparing the version of `KCSP` that features the advanced modelling (Section 5.2) with `KSAT`. The constraint solver that we use as the *sat* function in `KCSP` is based on search with chronological backtracking and constraint propagation. The propagation algorithms are specialised for their respective constraints and enforce generalised arc-consistency on them, as discussed in Section 5 above.

6.1 Test Environment

State of the Art. In the area of propositional satisfiability checking there is a large and rapidly expanding body of experimental knowledge; see, e.g., [9]. In contrast, empirical aspects of modal satisfiability checking have only recently drawn the attention of researchers. We now have a number of test sets, some of which have been evaluated extensively [4,14,10,17,16]. In addition, we also have a clear set of guidelines for performing empirical testing in the setting of modal logic [14,16]. Currently, there are three main test methodologies for modal satisfiability solvers, one based on hand-crafted formulas, the other two based on randomly generating problems. To understand on what kinds of problems a particular prover does or does not do well, it helps to work with test formulas whose meaning can (to some extent) be understood. For this reason we opted to carry out our tests using the Heuerding and Schwendimann (HS) test set [14], which was used at the TANCS '98 comparison of systems for non-classical logics [26].

The HS Test Set. The HS test set consists of several classes of formulas for $\mathcal{K}$, and other modal logics we do not consider here; e.g., some problem classes for $\mathcal{K}$ are based on the pigeon-hole principle (*ph*) and a two-colouring problem on polygons (*poly*). Each class is generated from a parametrised logical formula. This formula is either a $\mathcal{K}$-theorem, that is provable, or only $\mathcal{K}$-satisfiable, that is non-provable; consequently the generated class only contains either *p*rovable formulas or *n*on-provable formulas, and is labelled accordingly. The table at the right lists all such classes for $\mathcal{K}$.

*p*rovable	*n*on-provable
branch_p	*branch_n*
d4_p	*d4_n*
dum_p	*dum_n*
grz_p	*grz_n*
lin_p	*lin_n*
path_p	*path_n*
ph_p	*ph_n*
poly_p	*poly_n*
t4_p	*t4_n*

Some of these parametrised formulas are made harder by hiding their structure or adding extra pieces. The parameters allow for the creation of modal formulas, in the same class, of differing difficulty. The idea behind the parameter is that the difficulty of proving formulas in the same class should be exponential in the parameter. This kind of increase in difficulty will make differences in the speed of the machines used to run the benchmarks relatively insignificant.

Benchmark Methodology. The benchmark methodology is to test formulas from each class, *starting with the easiest instance*, until the provability status of a formula can not be correctly determined within 100 CPU seconds. The result from this class will then be the parameter of the largest formula that can be solved within this time limit. The parameter ranges from 1 to 21.

6.2 Implementation

Let us turn to details of our implementation of the `KCSP` algorithm. We used the constraint logic programming system ECL^{i}PS^e [27]. The HS formulas are first negated, reduced in CNF and then translated into the format of `KCSP`.

We add the following heuristics to `KCSP` with minimising constraints (in an attempt) to reduce the depth of the `KCSP` search tree: the value `u` is preferred for box formulas, and among them for positively occurring ones. Furthermore, the instantiation ordering of box formulas is along the increasing number of nested boxes in them: e.g., $x[\Box p]$ is instantiated before $x[\Box\Box p]$.

6.3 Assessment

In this subsection we provide an assessment of the contributions made by the various "aspects" provided by our advanced modelling.

Aspect 3: Partial Assignments by Constraints. Do minimising constraints make a difference in practice? To address this question, here we focus on the so-called *branch* formulas in the HS test set. It is worth noticing the relevance of *branch_n* for automated modal theorem proving: the class of non-provable branch formulas, *branch_n*, is recognized as the hardest class of "truly modal formulas" for today's modal theorem provers, cf. [16]. These are the so-called Halpern and Moses branching formulas that "have an exponentially large counter-model but no disjunction [...] and systems that will try to store the entire model at once will find these formulae even more difficult" [16].

Figure 2 plots the run times of `KCSP` (with and without minimising constraints) on *branch* formulas. Clearly, minimising constraints do make difference. The superiority of `KCSP` with minimising constraints over `KCSP` with total assignments is particularly evident in the case of *branch* formulas. `KCSP` with minimising constraints manages to solve 13 instances of *branch_n* and all 21 of *branch_p* (in less than 2 seconds), and without only 2 instances are solved, for both flavours.

Why is it so? To understand the reasons for the superiority of KCSP with minimising constraints, let us first consider what happens with *branch_p*(3), which KCSP with total assignments is already unable to solve (see Figure 2). In KCSP with minimising constraints, there are two choices for box formulas at layer 0 (i.e., with the same modal depth as *branch_p*(3)), and none at the subsequent layers of modal formulas obtained by "peeling off" one box from *branch_p*(3) (see Subsection 4.1). This results in a modal search tree of exactly two branches. Instead, with total assignments there are 6 extra box formulas at layer 0, which implies an extra branching factor of $2^6 = 64$ at the root of the modal search tree only. All 6 box formulas will always be carried over to subsequent layers, positively or negatively.

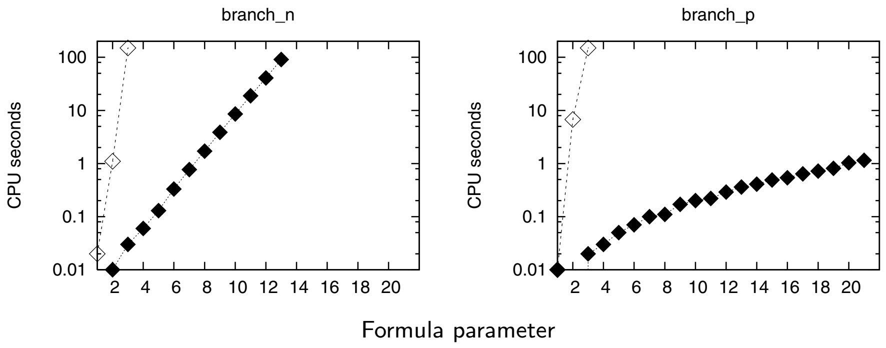


Fig. 2. KCSP with (♦) and without (◇) minimising constraints *branch* formulas. (Left): CPU time for $branch_n$ (seconds, log scale); (Right): $branch_p$ (seconds, log scale).

More in general, the superiority of KCSP with minimising constraints can be explained as follows: *the tree-like model that the solver (implicitly) attempts to construct while trying to satisfy a formula is kept as small as possible by the minimising constraints.* In this sense, constraints allows us to direct the modal search better than, for instance, unit propagation allows for in DPLL. We refer the reader to Subsection 6.4 below for more on this point.

Notice also that the results of KCSP with minimising constraints on the *branch* class are competitive with the best optimized modal theorem provers *SAT and DLP on the this class.

Aspect 4: Negated-CNF Constraint. In all the HS formula classes, having disjunctive constraints in place of CNF conversions increases the number of decided formulas, or, at least, does not decrease it. Avoiding CNF conversion by means of negated-CNF constraints may have a substantial effect, for example in the case of *ph_n*(4) — an instance of the pigeon-hole problem — which can now be solved in a few seconds. In contrast, by requiring CNF conversion (even with minimising constraints), ECL^{i}PS^e terminates the execution of KCSP pre-emptively for lack of memory. Unfortunately, the CNF conversion still necessary at the top level remains, and even prevents entering KCSP for formulas $ph(k)$ with $k > 4$.

Aspect 5: Factoring Constraint. This constraint avoids simple contradictory occurrences of a formula in the subsequent layer. We remark that this consideration of multiple occurrences of a subformula does not always provide a strictly minimal number of box formulas with a Boolean value. Nevertheless, it proved beneficial for formulas with the same variables hidden and repeated inside boxes. In fact, it proved useful in all of the following cases: *grz*, *d*4, *dum_p*, *path_p*, *t*4*p_p*. In the remaining cases the contribution of factoring with constraints is insignificant, except for *path_n* where searching for candidate formulas to be factored slightly slows down search.

Formula Simplifications. As a preprocess to KCSP, the top-level input formula may be simplified to a logically equivalent formula. We use standard simplification rules for propositional formulas, at all layers, in a bottom-up fashion. Also, in the same manner, the following modal equivalences are used in simplifying a CNF formula: $\neg\Box\top \wedge \psi \leftrightarrow \bot \wedge \psi$ and its dual. Simplification in KCSP plays a relevant role in the case of *lin* formulas. E.g., consider *lin_n*(3): without simplifications and minimising constraints, KCSP takes longer than 5 minutes to return an answer; by adding simplifications and minimising constraints, KCSP takes less than 0.4 seconds; besides, by also adding factoring, KCSP solves the most difficult formula of *lin_n* in 0.06 seconds, that of *lin_p* in 0.01.

6.4 Results and a Comparison

In this part, we compare the performances of KSATC and KCSP on the HS test-set; see Table 1 below. Each column in the table lists a formula class and the number of the most difficult formula decided within 100 CPU seconds by each prover; we write $>$ when all 21 formulas in the test set are solved within this time slot. We now explain the systems being compared in Table 1. The rows of the table are explained as follows.

First Row: KSATC. The results for KSATC (KSAT implemented in C++) are taken from [16]; there, KSATC was run with the HS test set on a 350 MHz Pentium II with 128 MB of main memory.

Second Row: KCSP. We used KCSP with all advanced aspects considered: i.e., partial assignments by constraints; negated-CNF constraints; factoring constraints; and formula simplifications. In the remainder, we refer to this as KCSP. The time taken by the translator from the HS format into that of KCSP is insignificant, the worst case among those in the comparison Table 1 taking less than 1 CPU second; these timings are included in the table entries for KCSP. We ran our experiments on a 1.2 GHz AMD Athlon Processor, with 512 MB RAM, under Red Hat Linux 8 and ECL^{i}PS^e 5.5.

Third Row: **KCSP/speed.** To account partially for the different platforms for KSATC and KCSP, we scaled the measured times of KCSP by a factor 350/1200, the ratio of the processor speeds. The results are given in the bottom line KCSP/speed, and emphasized where different from KCSP.

Table 1. The top two rows give the most difficult formula of each HS class decided by KSATC and KCSP, respectively, in 100 CPU/s; > means that all formulas in that class are decided. In the bottom row, KCSP/speed, the figures for KCSP are obtained after scaling the measured times by the ratio of the processor speeds of KSATC and KCSP.

	branch		d4		dum		grz		lin		path		ph		poly		t4p	
	n	p	n	p	n	p	n	p	n	p	n	p	n	p	n	p	n	p
KSATC	8	8	5	8	>	11	>	17	3	>	8	4	5	5	12	13	18	10
KCSP	13	>	6	9	19	12	>	13	>	>	11	4	4	4	16	10	7	10
KCSP/speed	*11*	>	6	*8*	*17*	*11*	>	*10*	>	>	*9*	4	4	4	16	*9*	*6*	*8*

Result Analysis. Note, first of all, that KSATC is compiled C++ code while KCSP is interpreted ECL^{i}PS^e (i.e., PROLOG); this makes it very interesting to see that the performance of KCSP is often competitive with that of KSATC. There are some interesting similarities and differences in performance between KSATC and KCSP. For some classes, KCSP clearly outperforms KSATC, for some it is the other way around, and for yet others the differences do not seem to be significant.

For instance, KCSP is superior in the case of *lin* and *branch* formulas. In particular, as pointed out in Subsection 6.3 above, *branch_n* is the hardest "truly modal test class" for the current modal provers, and KCSP with partial assignments performs very well on this class. Now, KSATC features partial assignments, just like KCSP does, in that its underlying propositional solver is DPLL. So, why such differences in performance? The differences are due to our modelling and the reasons for these can be explained as follows in more general terms:

– extended domains and constraints allows for *more control over the partial assignments* to be returned by the adopted constraint solver than unit propagation allows for in DPLL;
– constraints allow us to represent, in a very compact manner, certain requirements such as that of reducing the number of box formulas to which a Boolean value is assigned.

Consequently, the models that KCSP (implicitly) tries to generate when attempting to satisfy a formula remain very small. In particular, in the case of *branch*, searching for partial assignments with minimising constraints yields other benefits *per se*: the smaller the number of box formulas to which a Boolean value is assigned at the current layer, the smaller the number of propositions in the subsequent layer; in this manner fewer choice points and therefore fewer search tree branches are created. Thereby adding constraints to limit the number of box formulas to reason on, while still exploring the truly propositional search space, seems to be a winning idea on the *branch* class.

In the cases of *grz* and *t*4, instead, `KSATC` is superior to `KCSP`. Notice that `KSATC` features a number of optimisations for early modal pruning that are absent in `KCSP`, and these are likely to be responsible for the better behaviour of `KSATC` on these classes.

7 Finale

7.1 Looking Back

We described a constraint-based model for modal satisfiability. Thanks to this model, we could embed modal reasoning cleanly into pre-existing constraint programming systems, and directly make use of these to decide modal satisfiability. In this paper, we adopted the constraint logic programming system ECL^{i}PS^e.

In our base-model for modal satisfiability, we have extended domains for box formulas and appropriate constraints to reason about them; we also implemented (Section 5) and experimentally compared (Section 6) `KCSP` with further modelling constraints and heuristics for modal reasonings. In particular, `KCSP` with minimising constraints results to be competitive with the best modal theorem provers on the hardest "truly modal class" in the Heuerding and Schwendimann test set, namely *branch*; here, the addition of minimising constraints results in a significant reduction of the size and especially the branching of the "the tree-model" that our solver implicitly tries to construct for the input formula (Subsections 6.3 and 6.4).

More in general, an important advantage of our constraint-based modelling is that encoding optimisations (e.g., for factoring or partial assignments) can be done very elegantly and in an "economical" manner: that is, it is sufficient to add appropriate, compact constraints to obtain specific improvements (e.g., factoring or minimising constraints). Besides compactness in the models, extended domains and constraints allow for more control over the assignments to be returned by constraint solvers than unit propagation allows for in DPLL, as amply discussed in Subsection 6.4.

7.2 Looking Ahead

We conclude by elaborating on some open questions.

Modelling. Our current modelling of propositional formulas as finite CSPs can perhaps be enhanced so as to completely avoid CNF conversions. This could be achieved by (negated) clauses that may contain propositional formulas as well, not just variables or box formulas. The factoring constraints for controlling 0/1/`u` assignments to multiple occurrences of the same box formula are currently not optimally restrictive. Integrating them better with the assignment-minimising clause constraints can further reduce the modal spanning factor in certain situations. Also, we want to consider the literal modelling of propositions [28] and compare it with the one presented in this paper on modal formulas; in the literal modelling of ϕ, for each clause of *Prop*(ϕ) there is a variable domain containing the literals of *Prop*(ϕ); binary constraints would then be imposed among those domains that share a *Prop*(ϕ) variable of the form p or $x[\Box\phi]$. One advantage of this modelling is that partial Boolean assignments for *Prop*(ϕ) would come for free, i.e., without the need of enlarging the domains with an u value; yet, the control of the modal search procedure may require novel constraints.

Constraint Algorithm. Simple chronological backtracking may not be the optimal choice for the problem at hand. Efficiency can be expected to increase by remembering previously failed sub-propositions (nogood recording, intelligent backtracking), and also successfully solved sub-problems (lemma caching).

Logic. Many-valued modal logics [8] allow for propositional variables to have further values than the Boolean 0 and 1. Our approach to modal logics via constraint satisfaction can be easily and naturally extended to deal with finitely-valued modal logics.

Acknowledgements

We thank Juan Heguiabehere for supplying us with a translator from HS formulas into KCSP format. Rosella Gennari was supported by the post-doc project grant *Automated Reasoning by Constraint Satisfaction* from the Autonomous Province of Trento. Maarten de Rijke was supported by grants from the Netherlands Organization for Scientific Research (NWO), under project numbers 612-13-001, 365-20-005, 612.069.006, 612.000.106, 220-80-001, and 612.000.207.

References

1. K.R. Apt and S. Brand. Schedulers for rule-based constraint programming. In *Proc. of ACM Symposium on Applied Computing (SAC'03)*, pages 14–21. ACM, 2003.
2. C. Areces, R. Gennari, J. Heguiabehere, and M. de Rijke. Tree-based Heuristics in Modal Theorem Proving. In *Proc. of the 14th European Conference on Artificial Intelligence 2000*, pages 199–203. IOS Press, 2000.
3. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook.* Cambridge University Press, 2003.
4. F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.-J. Profitlich. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or: Making KRIS get a move on. In *Proc. KR-92*, pages 270–281, 1992.
5. F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116, 2000.
6. P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic.* Cambridge University Press, 2001.
7. L. Cardelli and A.D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In *Proc. of the 27th ACM Symposium on Principles of Programming Languages*, 2000.

8. M.C. Fitting. Many-valued modal logics II. *Fundamenta Informaticae*, XVII:55–74, 1992.
9. I. Gent, H. Van Maaren, and T. Walsh, editors. *SAT 2000.* IOS Press, 2000.
10. F. Giunchiglia and R. Sebastiani. Building Decision Procedures for Modal Logics from Propositional Decision Procedures. The Case Study of Modal $\mathbf{K}(m)$. *Information and Computation*, 162(1–2):158–178, 2000.
11. V. Haarslev and R. Möller. RACER. Accessed via `http://kogs-www.informatik.uni-hamburg.de/~race/`, September 2002.
12. J.Y. Halpern, R. Harper, N. Immerman, P.G. Kolaitis, M.Y. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic*, 7:213–236, 2001.
13. B.P. Harrenstein, W. van der Hoek, J.-J.Ch. Meyer, and C. Witteveen. On modal logic interpretations of games. In *Proc. of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, 2002.
14. A. Heuerding and S. Schwendimann. A Benchmark Method for the Propositional Modal Logics **K**, **KT**, **S4**. Technical Report IAM-96-015, University of Bern, 1996.
15. I. Horrocks. FaCT. Accessed via `http://www.cs.man.ac.uk/~horrocks/FaCT/`, September 2002.
16. I. Horrocks, P.F. Patel-Schneider, and R. Sebastiani. An Analysis of Empirical Testing for Modal Decision Procedures. *Logic Journal of the IGPL*, 8(3):293–323, 2000.
17. U. Hustadt and R.A. Schmidt. On Evaluating Decision Procedures for Modal Logic. In *Proc. IJCAI-97*, pages 202–207, 1997.
18. M.R.A. Huth and M.D. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, 1999.
19. M. Marx. XPath with conditional axis relations. In *Proc. of the International Conference on Extending Database Technology*, 2004.
20. MSPASS V 1.0.0t.1.2.a. Accessed via `http://www.cs.man.ac.uk/~schmidt/mspass`, 2001.
21. G. Pan, U. Sattler, and M.Y. Vardi. BDD-Based Decision Procedures for **K**. In *Proc. of CADE 2002*, pages 16–30. Springer LINK, 2002.
22. P.F. Patel-Schneider. DLP. Accessed via `http://www.bell-labs.com.user/pfps/dlp/`, September 2002.
23. M. Pistore and P. Traverso. Planning as Model Checking for Extended Goals in Non-Deterministic Domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.
24. A.S. Rao and M.P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8:293–342, 1998.
25. A. Tacchella. *SAT System Description. In *Collected Papers from the International Description Logics Workshop 1999, CEUR*, 1999.
26. TANCS: Tableaux Non-Classical Systems Comparison. Accessed via `http://www.dis.uniroma1.it/~tancs`, 2000.
27. M.G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.
28. T. Walsh. SAT v CSP. In R. Dechter, editor, *Proc. of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 441–456, 2000.

Learning Approximate Consistencies

Arnaud Lallouet, Andreï Legtchenko,
Thi-Bich-Hanh Dao, and AbdelAli Ed-Dbali

Université d'Orléans, LIFO
BP 6759, F-45067 Orléans, France

Abstract. In this paper, we present an abstract framework for learning a finite domain constraint solver modeled by a set of operators enforcing a consistency. The behavior of the consistency to be learned is taken as the set of examples on which the learning process is applied. The best possible expression of this operator in a given language is then searched. We present sufficient conditions for the learned solver to be correct and complete with respect to the original constraint. We instantiate this framework to the learning of bound-consistency in the indexical language of Gnu-Prolog.

1 Introduction

1.1 Motivation

Constraint Satisfaction Problems (or CSPs) have been widely recognized as a powerful tool to model, formulate and solve artificial intelligence problems as well as industrial ones. A common framework to address this task is the combination of search and filtering by a local consistency. Enforcing a local consistency is usually done in constraint solvers by the scheduling of monotonic and contracting operators up to reaching their greatest common fixpoint. For simple arithmetic constraint, these operators are relatively easy to devise but for more complex ones such as global constraints, months of work are necessary to produce a good filtering procedure.

Between these two extremes lie a population of intermediate constraints which deserve a special treatment. They are usually either decomposed in a conjunction of small basic constraints or expressed by a table. When decomposed, variable decorrelation may alter their pruning power and the multiplicity of smaller constraints adds many operators to the scheduling process. When expressed by a table, the classical arc-consistency algorithm involves many traversal of the table which considerably slows down the solving process.

Our contribution consists in finding automatically a set of operators dedicated to this particular type of constraint and which follows as much as possible the behavior of a given level of consistency. These operators are directly included in the propagation queue of the solver in replacement of the ones generated by the original constraint or set of constraints. A tool for automatic solver construction is likely to be part of an environment for Constraint Programming. It can help the fine tuning of an application by providing for free a stronger propagator for a group of constraints or an ad-hoc global constraint. It may also be useful to the solver designer to provide a first implementation of a new constraint. This paper is an extended and revised version of [15].

1.2 Overview

A consistency is often thought of as a property which must hold for a constraint network. Usually, this property is local and can be expressed by considering only a few number of constraints at a time (typically one or two). The use of monotonic and contracting operators to enforce a consistency is well-known: the consistency is obtained as the common closure of the set of operators. This closure is efficiently computed by a chaotic iteration of the operators [3]. But usually, the task of finding efficient operators which actually define a consistency is considered as one of the smartest skills of the solver's implementor. The essence of this learning framework consists in considering the behavior of the operator enforcing the desired consistency as a set of examples in order to find an adequate representation of this operator in a given language. Then we explore the space defined by this language in order to find the best possible representation of the expected consistency.

Finding a good consistency operator means that it should be *contracting* to reduce the search space and it should be *monotonic* in order to ensure the confluence of the reduction mechanism. But it should also be *correct* with respect to the constraint. It means that we need to ensure that it never rejects a solution of a constraint at any state of the search space. We present a technique to deal with this aspect without having to check the whole search space. Finally, the learned operator should *represent* the constraint. It means that it should act a satisfiability check which accepts a solution tuple and rejects all the others. This is the basis of the translation of constraints into operators: a constraint is replaced by an operator which preserves its original semantics while having an active behavior in the search space. We present an original method which consists in repairing an incomplete solver in order to get this property. Although operators could be expressed in virtually any language, the use of the so-called glass-box approach of indexicals [23, 13] proves again the qualities of a declarative language.

The rest of the paper is organized as follows. In section 2, we introduce a set-theoretic and language-independent framework which allows to define consistencies with their properties, to compare them and a criterion to find a representation in any language. In section 3, we instantiate this framework to the language of indexicals for the construction of operators for bound-consistency, we describe our implementation, called the Solar system (Solar stands for SOlver LeARning) and the integration of the produced solver in Gnu-Prolog [13]. We end by a few representative examples and some benchmarks.

2 Theoretical Framework

Let V be a set of variables and $D = (D_X)_{X \in V}$ their (finite) domains. For $W \subseteq V$, we denote by D^W the set of tuples on W, namely $\Pi_{X \in W} D_X$. Projection of a tuple or a set of tuples on a variable or a set of variables is denoted by $|$, natural join of two sets of tuples is denoted by $\bowtie$. If A is a set, then $\mathcal{P}(A)$ denotes its powerset.

Definition 1 (Constraint). *A* constraint *c is a pair (W, T) where:*

- *$W \subseteq V$ is the* arity *of the constraint c and is denoted by $var(c)$.*
- *$T \subseteq D^W$ is the set of* solutions *of c and is denoted by $sol(c)$.*

The join of two constraints is defined as a natural extension of the join of tuples: the *join* of c and c' is the constraint $c \bowtie c' = (var(c) \cup var(c'), sol(c) \bowtie sol(c'))$.

Definition 2 (CSP). *A* CSP *is a set of constraints.*

Join is naturally extended to CSPs and the *solutions* of a CSP C are $sol(\bowtie C)$. A direct computation of this join is too expensive to be tractable. This is why a framework based on approximations is preferred, the most successful of them being the domain reduction scheme where variable domains are the only reduced constraints (see [3] for a more general framework). So a search state consists in a set of yet possible values for each variable.

Definition 3 (Search state and search space). *For $W \subseteq V$, a* search state *is a family $s = (s_X)_{X \in W}$ such that $\forall X \in W, s_X \subseteq D_X$. The corresponding* search space *is $S_W = \Pi_{X \in W} \mathcal{P}(D_X)$.*

The set S_W, ordered by pointwise inclusion $\subseteq$ is a complete lattice. Some search states we call *singletonic* play a special role in our framework. A singletonic search state comprises a single value for each variable, and hence represents a single tuple. A tuple is promoted to a singletonic search state by the operator $\lceil\ \rceil$:

$$\lceil\ \rceil : D^W \to S_W \text{ where for } t \in D^W, \lceil t \rceil = (\{t_X\})_{X \in W} \in S_W$$

This notation is extended to a set of tuples: for $E \subseteq D^W$, let $\lceil E \rceil = \{\lceil t \rceil \mid t \in E\} \subseteq S_W$. Conversely, a search state is converted into the set of tuples it represents by taking its cartesian product Π : for $s \in S_W$, $\Pi s = \Pi_{X \in W} s_X \subseteq D^W$. We denote by $Sing_W$ the set $\lceil D^W \rceil$ of singletonic search states. By definition, $\lceil D^W \rceil \subseteq S_W$.

2.1 Consistency

A consistency for a constraint c is an operator on S_W having some properties. In order to set our learning framework where any kind of operator is present in the hypothesis space, we need to isolate each property independently. Let $W \subseteq V$ and let f be an operator on S_W:

- f is *monotonic* if $\forall s, s' \in S_W, s \subseteq s' \Rightarrow f(s) \subseteq f(s')$.
- f is *contracting* if $\forall s \in S_W, f(s) \subseteq s$.
- we denote by $Fix(f)$ the set of fixpoints of f. They define the *consistent states* according to f.

– for $W \subseteq W' \subseteq V$, an operator f on S_W can be *extended* to f' on $S_{W'}$ by taking: $\forall s \in S_{W'}, f'(s) = s'$ with $\forall X \in W' \backslash W, s'_X = s_X$ and $\forall X \in W, s'_X = f(s|_W)_X$. Then $s \in Fix(f') \Leftrightarrow s|_W \in Fix(f)$.

An operator on S_W is *associated* to a constraint $c = (W, T)$ if its singletonic fixpoints represent the solution tuples T:

Definition 4. *An operator $f : S_W \rightarrow S_W$ is* associated *to a constraint $c = (W, T)$ if:*

$$Fix(f) \cap Sing_W = \lceil T \rceil$$

However, nothing is said about its behavior on non-singletonic states. But, in order for an operator to be a good candidate to be a consistency, we need to ensure that the operator is monotonic, contracting and correct, meaning that no solution tuple could be rejected at any time anywhere in the search space. We call such an operator a *pre-consistency*:

Definition 5 (Pre-consistency). *An operator $f : S_W \rightarrow S_W$ is a* pre-consistency *for $c = (W, T)$ if:*

– *f is monotonic.*
– *f is contracting.*
– *$\forall s \in S_W, \Pi s \cap T \subseteq \Pi f(s) \cap T$.*

Since a pre-consistency is contracting, the above inclusion which is required not to loose any solution is actually an equality. Note that a pre-consistency is not obligatorily associated to its constraint since the set of its singletonic fixpoints may be larger. When it coincides, we call such an operator a *consistency*:

Definition 6 (Consistency). *An operator associated to c and which is also a pre-consistency for it is called a* consistency *for c.*

With this property on singletonic states, the consistency check for a candidate tuple can be done by the propagation mechanism itself. Note that a consistency is an extension to S_W of the satisfiability test made on singletonic states.

Example 7. Consider the constraint c given by $Y = X \pm 2$ with $D_X = [0, 10]$, $D_Y = [0, 8]$ and depicted by the encircled dots in figure 1. The following operators:

```
X in min(Y)-2 .. max(Y)+2
Y in min(X)-2 .. max(X)+2
```

define a pre-consistency for c since they are correct with respect to c: they do not reject a tuple which belongs to the constraint. For example, the dark grey area shows the evaluation of the first operator for X when $Y \in [3..6]$. However, the operators are not associated to c since they accept more tuples than necessary, like for example $(4, 4)$ indicated by the black square. Actually, they accept the light grey area and are a consistency for the constraint $X - 2 \leq Y \leq X + 2$.

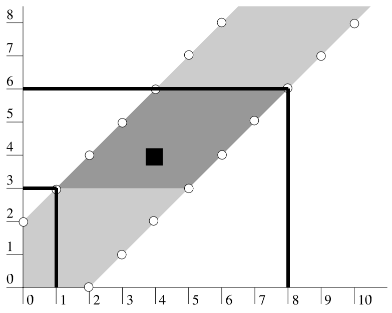


Fig. 1. Preconsistency for $Y = X \pm 2$

Let us now define some consistencies associated to a constraint $c = (W, T)$:

- ID_c is a family of contracting operators such that any id_c in ID_c verifies: $\forall s \in S_W \backslash Sing_W$, $id_c(s) = s$ and $\forall s \in Sing_W, s \in \lceil T \rceil \Leftrightarrow id_c(s) = s$. In particular, on non-solution singletonic states, id_c reduces at least one variable's domain to $\emptyset$. This comes from the fact that all search states s such that $\Pi s = \emptyset$ represent the empty set of solution for a constraint. In the following, we denote by id_c any member of ID_c.
- ac_c is the well-known arc-consistency operator defined by $\forall s \in S_W, ac_c(s) = s'$ with $\forall X \in W$, $s'_X = (\Pi s \cap T)|_X$.

We now suppose that each variable domain D_X is equipped with a total ordering $\leq$. The notation $[a..b]$ is used for the classical notion of interval $\{e \in D_X \mid a \leq e \leq b\}$. We call Int_X the interval lattice built on D_X and $Int_W = \Pi_{X \in W} Int_X$. For $A \subseteq D_X$, we denote by $[A]$ the set $[\min(A)..\max(A)]$. The bound-consistency consists in contracting only the bounds of a variable's domain:

- bc_c : is the bound-consistency operator defined by $\forall s \in S_W, bc_c(s) = s'$ with $\forall X \in W$, $s'_X = s_X \cap [(\Pi s \cap T)|_X]$

Note that bound-consistency can be freely mixed with other consistencies which remove values inside a variable's domain (like arc-consistency) because the interval computed by bc_c is intersected with the original domain in order to preserve the anterior removal of values inside the domain.

Monotonic contracting operators have another interesting property: they preserve the solution tuples of their associated constraint across the search space. This is the central property we use to construct preconsistencies:

Proposition 8. *Let f be a monotonic and contracting operator such that $\lceil sol(c) \rceil \subseteq Fix(f)$. Then f is a pre-consistency for c.*

Proof. Suppose that f is not a pre-consistency. Then $\exists s \in S$ and $t \in sol(c)$ such that $t \in \Pi s$ and $t \notin \Pi f(s)$, or equivalently $\lceil t \rceil \subseteq s$ and $\lceil t \rceil \not\subseteq f(s)$. Since $t \in sol(c)$ and $\lceil sol(c) \rceil \subseteq Fix(f)$, we have $\{t\} = \Pi f(\lceil t \rceil)$. It yields that $\lceil t \rceil \subseteq f(\lceil t \rceil)$ and $\lceil t \rceil \not\subseteq f(s)$. Hence we have $\lceil t \rceil \subseteq s$ and $f(\lceil t \rceil) \not\subseteq f(s)$, which contradicts the fact that f is monotonic.

In practice, Proposition 8 allows to reduce considerably the search space which must be checked to verify if an operator is a pre-consistency for a constraint c. It is sufficient to verify if the operator preserves the solutions of c when expressed as singletonic search states. For more general operators, in particular non-monotonic ones, we should make this verification for every state of S_W.

Let $F = \{f_1, \ldots, f_n\}$ be a set of consistencies on S_W associated respectively to $\{c_1, \ldots, c_n\}$ and F' be a set of pre-consistencies for $\{c_1, \ldots, c_n\}$. If all constraints are not defined on the same set of variables, it is always possible to use the extension of the operators on the union of all variables which appear in the constraints. The common closure of the operators of $F \cup F'$ can be computed by a chaotic iteration [3]. It follows from the main confluence theorem of chaotic iterations that a consistency can be constituted by combining the mutual strengths of several operators. Since we have $Fix(F \cup F') = \bigcap_{f \in F \cup F'} Fix(f)$ and since each operator does preserve the tuples of its associated constraint, the computed closure of all operators associated to the CSP does not reject a tuple of $c_1 \bowtie \ldots \bowtie c_n$ for any $s \in S_W$ because of an operator of F.

We will use this property in our instantiation of the framework to construct a consistency for a constraint c. We learn a *pre-consistency* l instead of a consistency because enforcing simultaneously correction *and* completeness on singleton states is difficult, and even not always possible within a given language. Indeed, for example, the indexical language we use in the next section has been tailored for the classical built-in constraints and becomes cumbersome for ad-hoc constraints. Then, a full consistency is reconstituted by combining the consistency id_c which is associated to c but has the weakest reduction strength with the learned pre-consistency l for which we do not care about the rejection of each non-solution tuple but which tries to get as close as possible to the desired consistency along the whole search space. This combination allows to *repair* the learned pre-consistency in order to get completeness.

2.2 Approximation and Learning Framework

Since our purpose is to find automatically an operator closest to a given consistency, we need to provide a comparison criterion:

Definition 9 (Precision ordering). *Let f and f' be two operators on S_W and let $E \subseteq S_W$. We say that f is* more precise *than f' on E, denoted by $f \subseteq_E f'$ if $\forall s \in E, f(s) \subseteq f'(s)$. If $E = S_W$, we simply write $f \subseteq f'$.*

Of course this is a partial ordering and some consistencies may be uncomparable. But they are always less precise than arc-consistency, as stated by the following proposition:

Proposition 10. *The consistency ac_c is the most precise consistency for c.*

Proof. Suppose that there exists a consistency f for c such that $f \subset ac_c$. Since f is contracting, $\exists s \in S, \exists X \in W, \exists v_X \in D_X$ such that $v_X \in ac_c(s)_X$ and $v_X \notin f(s)_X$. But $ac_c(s)_X = s_X \cap sol(c)_X$. Hence $\exists t \in sol(c)$ such that $t_X = v_X$

and $\lceil t \rceil \subseteq s$. But, since f is a pre-consistency, we have $f(\lceil t \rceil) = \lceil t \rceil$ and in particular $f(\lceil t \rceil)_X = \{v_X\}$. But since $v_X \notin f(s)_X$, this contradicts the fact that f is monotonic.

Arc-consistency is the most precise consistency which considers only one constraint at a time. More pruning can be obtained by considering two or more constraints together at the same time (path- and k-consistency for example, see [11]).

When $f \subseteq_E f'$, we say that f' is *correct* w.r.t. f on E. It means that f' rejects less tuples than f. In particular, if f is a pre-consistency for c, so is f'. Conversely, if $f \subseteq f'$, we say that f is *complete* w.r.t. f' on E. If $f \subseteq_{Sing_W} f'$, we say that f is *singleton complete* w.r.t. f'. The following notion of approximation defines the objects we want to build. If $cons_c$ is the consistency to be learned, we want to build an operator f which is correct and defines exactly the same constraint as $cons_c$:

Definition 11 (Approximation). *Let f and f' be two pre-consistencies on S_W. The pre-consistency f'* approximates *f if $f \subseteq f'$ and $f' \subseteq_{Sing_W} f$.*

For example, bc_c is an approximation of ac_c.

In order to find the best possible expression for an operator in a given language, we have to be able to compare two operators. This is usually done with a distance. Let d be such a distance between two operators. Let $\mathcal{L}$ be a language in which the operators are expressed and let $[\![.]\!]$ be an interpretation for this language such that every term $l \in \mathcal{L}$ evaluates into an operator $[\![l]\!]$. The problem of learning an operator for a consistency $cons_c$ amounts in finding a term $l_0 \in \mathcal{L}$ such that $d(cons_c, [\![l_0]\!]) = \min_{l \in \mathcal{L}} d(cons_c, [\![l]\!])$. But only minimizing this distance does not ensure that the learned operator is actually an approximation of $cons_c$. We must also ensure the correctness of $[\![l_0]\!]$ w.r.t. $cons_c$. Hence the learning problem becomes:

Definition 12 (Learning Problem).

$$\begin{array}{rl} \texttt{minimize} & d(cons_c, [\![l]\!]) \\ \texttt{subject to} & \forall s \in S_W, cons_c(s) \subseteq [\![l]\!](s) \end{array}$$

The correctness of the method is ensured by construction. Following machine learning vocabulary, $cons_c$ represents the *example* space and $\mathcal{L}$ the *hypothesis* space.

By Proposition 8, we can reduce the constraint part to $\lceil sol(c) \rceil$ because the operator we want to build is monotonic. Actually, we take a superset E of $\lceil sol(c) \rceil$ for the learning task in order to have a good convergence.

3 Learning Indexicals

In this section, we instantiate our theoretical framework. Since finding an arithmetic expression is a notoriously difficult task, we had to define strong language biases in order to limit the combinatorial explosion.

c ::=	X in r	
r ::=	t_1 .. t_2	interval
	{ t }	singleton
	r_1 : r_2	union
	r_1 & r_2	intersection
t ::=	`min(Y)`	indexical *min*
	`max(Y)`	indexical *max*
	`val(Y)`	indexical *val*
	ct	constant term
	$t_1+t_2 \mid t_1-t_2 \mid t_1*t_2 \mid$	int ops
	t_1 /< t_2 $\mid$ t_1 /> t_2	
ct ::=	n $\mid$ `infinity` $\mid$	values
	$ct_1+ct_2 \mid ct_1-ct_2 \mid ct_1*ct_2 \mid$	
	ct_1 /< ct_2 $\mid$ ct_1 /> ct_2	

Fig. 2. A subset of Gnu-Prolog indexical language

3.1 Gnu-Prolog Indexical Language

Indexicals have been introduced in [23] in order to define finite domains constraint solvers and have been used in several powerful implementations of CLP systems such as Gnu-Prolog [13] or Sicstus-Prolog [8]. In this language, an operator is written "`X in r`" in which `X` actually represents the domain of the variable X and `r`, called the range, is an expression representing a subset of D_X. The range is an expression which possibly involves references to other variables' current domain such as `min(Y)` or `max(Z)`. If we denote by x the unary constraint representing X's domain, the above indexical can be read declaratively as the second-order constraint $x \subseteq r$ or operationally as the operator $x \mapsto x \cap r$. High-level constraints are compiled into `X in r` operators. Here is an example for the boolean `and` constraint:

Example 13. The constraint "$X \wedge Y = Z$", where X, Y and Z are booleans (0 or 1), under bound-consistency is compiled into:

```
X in min(Z) .. max(Z)*max(Y)+1-min(Y)
Y in min(Z) .. max(Z)*max(X)+1-min(X)
Z in min(X)*min(Y) .. max(X)*max(Y)
```

A first remark is that the first two expressions are not very intuitive. Building such expressions by hand for complex constraints is certainly a challenge. Moreover, the compilation scheme is predefined only for some (at most ternary) constraints like arithmetic or boolean constraints, which are directly compiled. The others are internally decomposed with the use of hidden variables. While the declarative meaning is identical (the constraint and its decomposition have the same set of solutions), these expressions are not treated globally and this weakens their pruning power. However, for the built-in constraints, the set of produced indexicals defines exactly the expected consistency (bound or arc) and is singleton complete. A subset of the Gnu-Prolog indexical language is given in figure 2. Note that the indexical `val(X)` has a special behavior: it contains a guard which delays the evaluation of the operator until X is instantiated.

Choice 14. We use the indexical language as target language to express the learned operators.

By this choice, we will have to learn at least one indexical per variable. This choice is motivated by the ease of integration of the user-defined indexicals in Gnu-Prolog. Indeed, user-defined indexicals can be placed in a separate file with extension `.fd`, which is compiled and linked with the Gnu-Prolog program which uses them [13]. Here is an example of the syntax of an `.fd` file for the and constraint, where the declaration `fdv X` is used to link the variable `X` between the Gnu-Prolog core and the constraint definition file:

```
and(fdv X, fdv Y, fdv Z)
{
 start X in min(Z) .. max(Z)*max(Y)+1-min(Y)
 start Y in min(Z) .. max(Z)*max(X)+1-min(X)
 start Z in min(X)*min(Y) .. max(X)*max(Y)
}
```

3.2 Construction of a Consistency

The bound-consistency allows to limit the example space to intervals instead of arbitrary subsets. If d is the size of a variable's domain and n the arity of the constraint, its size boils down to $(d^2/2)^n$ instead of 2^{nd} with subsets.

Choice 15. We learn bound-consistency: the closure of learned operators approximates the bound-consistency.

The example set to learn an operator for bound-consistency for a variable X consists in the bounding box of the projection on X of every possible box of Int_W. This allows to learn each projection separately:

Definition 16 (Example set for bound-consistency bc_c). *Let $c = (W, T)$ be a constraint and let $X \in W$. The example set for the projection on X of bc_c is the function $bc_c^X : Int_W \to Int_X$ defined by:*

$$bc_c^X(s) = [(\Pi s \cap T)|_X]$$

Different consistencies are mixable within the framework of chaotic iterations, thus the operator bc_c^X could take its argument in S_W instead of Int_W. It would allow to preserve the values removed inside the domains but at the price of a considerable increase in the size of the example space. And since the operators we want to build only concern the bounds of the intervals, considering the removal of values inside them would not lead to any improvement in the quality of the bounds found. Thus this definition is justified in this learning framework. Since here s is assumed to belong to Int_W, there is also no need to intersect with s_X.

Since operators are scheduled in chaotic iterations, a consistency for a constraint can be constructed from several sub-operators which combine their mutual strengths toward a common goal. We choose to learn one indexical operator l_X per variable of the constraint and we only force this operator to be a preconsistency less precise than bc_c^X, i.e. $bc_c^X \subseteq l_X$. Then it is enough to compose the indexicals l_X $(X \in W)$ with id_c within a chaotic iteration to get a consistency for c. We call the indexicals for id_c *reparation* indexicals and the indexicals for l_X *reduction* indexicals.

Choice 17. For each constraint, we build one reparation operator and we learn one reduction indexical per variable.

The separation in two groups (reduction and reparation) is a good compromise to achieve the efficiency needed in the constraint propagation phase and the completeness also needed by the compilation scheme. However, the language of indexicals allows to express correct and complete operators for every constraint (see method 2 in section 3.4). But these operators are too slow when scheduled in the propagation queue to be useful, because thay are called a lot of times to reduce the variables' domains, though they may be acceptable as reparation ones. Moreover, the formulation of the learning problem as an optimization problem (definition 12) does not ensure that singleton completeness is achieved, even if possible. By delaying this problem to reparation operators, we can concentrate on building fast, simple and in average efficient operators for reduction. Note that this problem does not occur for the built-in constraints. They are regular enough for the indexicals language to capture correctness and completeness (and not only singleton completeness!) for both arc- and bound-consistency within a small, fast and simple expression.

3.3 Learning Process

In the next subsection, we will show three different implementations of the reparation operator. Let us first consider the reduction indexicals. The language as described in figure 2 allows to build many complex expressions for intervals, for example with union or intersection. We then have to face a huge hypothesis space. In order to reduce the complexity, we first have to limit the indexical form to a single interval, with bound functions for min and max:

Choice 18. The reduction indexical l_X for X is of the form `X in minl`$_X$ `..` `maxl`$_X$

Each bound can be learned separately. Hence, we have two separate example sets, one for each bound. For example, for the `min` bound, we have the function $minbc_c^X : s \mapsto \min(bc_c^X(s))$.

Nevetherless the form of `minl`$_X$ and `maxl`$_X$ is still not limited. Indeed, the indexical language allows to generate any integer function. Searching for the right expression in such a large space is known to be a very complex problem. Thus we also need to limit to some function families.

Choice 19. The bound functions $\texttt{minl}_X$ and $\texttt{maxl}_X$ belong to predefined function families.

In practice, we use linear, piecewise linear and rational bounds:

– linear:
$$a + \sum_{Y \in W \setminus \{X\}} (b_Y . \min(Y) + c_Y . \max(Y))$$

– piecewise linear with k pieces:
$$a + \sum_{1 \leq i \leq k} \sum_{Y \in W \setminus \{X\}} (b_{Y,i} .(c_{Y,i} \geq \min(Y)) + d_{Y,i} .(e_{Y,i} \geq \max(Y)))$$

– rational:
$$a + \sum_{Y \in W \setminus \{X\}} \left(\frac{b_Y}{\min(Y) + 1} + \frac{c_Y}{\max(Y) + 1} \right)$$

With such a fixed form, the learning problem amounts to find the best coefficients for the template expression. In order for the operator to be monotonic, we impose additional syntactic restrictions: the bound $\texttt{minl}_X$ must be anti-monotonic and $\texttt{maxl}_X$ monotonic. This can be ensured by carefully choosing the coefficient for each expression. For example in the linear case for the $\texttt{min}l_X$ bound, we must have $b_Y > 0$ and $c_Y < 0$. Similar syntactical conditions can be found for the other forms. A finer analysis based on a kind of abstract interpretation is proposed in [7] and is implemented in Sicstus-Prolog.

After that, we need to instantiate the distance we use to compare two operators, namely the candidate to be learned and the example consistency. As distance between two consistencies f_1 and f_2, we use the global error on the example space E:

$$\sum_{s \in E} |\Pi f_1(s) \setminus \Pi f_2(s) \ \cup \Pi f_2(s) \setminus \Pi f_1(s)|$$

Since we force the learned operator l_X to be correct with respect to the example consistency bc_c^X, we can use only the difference in one direction. Hence:

$$d(bc_c^X, l_X) = \sum_{s \in E} |l_X(s) \setminus bc_c^X(s)|$$

When interpreting this condition to the bound functions, the distance for $\min l_X$ is the following because correctness forces $\texttt{minl}_X$ to be less or equal than $minbc_c^X$:

$$d(minbc_c^X, \texttt{minl}_X) = \sum_{s \in E} minbc_c^X(s) - \texttt{minl}_X(s)$$

Conversely, for $\texttt{maxl}_X$, it yields:

$$d(maxbc_c^X, \texttt{maxl}_X) = \sum_{s \in E} \texttt{maxl}_X(s) - maxbc_c^X(s)$$

In order to ensure the best possible quality for an operator, it is highly desirable to use an example set as large as possible. Thus we can get the best approximation of the operator's genuine quality. But the size of this space may be huge, especially for n-ary ($n > 2$) constraints over large domains. The computation time of the distance function is directly proportional to the size of the example space, and thus so is the learning time. By using proposition 8, we can reduce the example set up to $\lceil sol(c) \rceil$.

Choice 20. We learn monotonic operators from the restriction of bc_c to a set $E_S \subseteq S_W$ such that $\lceil sol(c) \rceil \subseteq E_S$.

However, some additional states belonging to E and not to $\lceil sol(c) \rceil$ are nevertheless useful to improve the quality of the learned operator. Otherwise any monotonic operator would be accepted between bc_c and id_c, possibly close to id_c. In practice this worst case happens all the time. This is why we build E by adding to $\lceil sol(c) \rceil$ a few dozen of random points of S_W, with a maximum of a few hundreds when the search space is large. This parameter determines relatively the ratio between the learning time and the probable quality of the learned operator but quickly comes to an asymptote: on $\lceil sol(c) \rceil$, the learned operator is close to id_c, with a few dozen more points, it becomes acceptable but does not improve a lot when adding more points.

3.4 Construction of Reparation Operators

The reparation operator id_c can be implemented in three forms:

Method 1. When the constraint we are interested in can be expressed as a combination of predefined constraints, we can reuse the Gnu-Prolog indexicals for each individual constraint in which each instance of `min` and `max` is simply replaced by `val`. Since `val` delays the evaluation of the operator until instantiation of the variable, the transformed operators perform only the satisfiability test for the constraint. For example, the first operator of the `and` constraint given in example 13 becomes:

```
X in val(Z) .. val(Z)*val(Y)+1-val(Y)
```

Moreover, Gnu-Prolog detects this kind of operators and does not schedule them in the same propagation queue. Thus these satisfiability operators can be added at no cost during the resolution. This method is used in example 21 for the combination of linear constraints: each constraint is compiled by Gnu-Prolog and generates indexicals which are simply reused in their transformed form for our reparation operator.

Method 2. When the constraint is expressed by a finite set of tuples in a table, the indexical language still has a sufficient expressivity to express the reparation operator. By using integer division, the function $\dfrac{1}{1+(X-a)^2}$ is equal to

1 if $X = a$ and 0 otherwise. By combining such functions, we can address each tuple separately. The size of these expressions may be quite large (actually linear with respect to the size of the table) but can be automatically generated. In our examples, we eventually produced indexicals whose expression was over 30 kB of text. The successful integration of such large operators is a witness of Gnu-Prolog's robustness since no built-in constraint does have such a complexity. Let us describe a smaller example using a constraint $foo(X, Y)$ defined by $\{(2, 3), (3, 5)\}$. In order to test its satisfiability, we use an auxiliary variable R corresponding to the reification of foo, we add the constraint $R > 0$ and we generate the following indexical:

$$\text{R in } \left\{\frac{\texttt{1}}{\texttt{1} + (\texttt{val(X)} - \texttt{2})^2} * \frac{\texttt{1}}{\texttt{1} + (\texttt{val(Y)} - \texttt{3})^2} + \frac{\texttt{1}}{\texttt{1} + (\texttt{val(X)} - \texttt{3})^2} * \frac{\texttt{1}}{\texttt{1} + (\texttt{val(Y)} - \texttt{5})^2}\right\}$$

The domain of R is reduced to $\{\texttt{0}\}$ if the tuple does not belong to the constraint and to $\{\texttt{1}\}$ otherwise. This indexical and the constraint $R > 0$ express the id_c operator.

Method 3. A tuple can be addressed more naturally by using integer conditions combined by $=$, *or* and *and*. The indexicals language integrated in Gnu-Prolog and described in figure 2 does not contain these operators but Gnu-Prolog accepts user-defined indexicals in C language [12]. These indexicals are placed in `.fd` files, which are compiled and linked with the Gnu-Prolog main program. This way, we can use C operators like "==, &&, ||" to address tuples. For example, for the constraint foo we produce the following reparation indexical which will be used in conjunction with the constraint $R > 0$:

```
start R in {val(X)==2&&val(Y)==3 || val(X)==3&&val(Y)==5}
```

In practice, we use this method to build reparation operators for constraints which are defined by finite sets of tuples. Since they are directly written in C and compiled, they are faster than the other reparation methods and provide the best results. Of course, the generation of indexicals following this method is fully automatized.

3.5 Learning System

An implementation of this framework is done within our "Solar" system. From the set of solutions of the constraint, the system generates the reduction indexicals (one per variable). For the reparation indexicals, if the constraint can be expressed as a combination of predefined constraints, we use method 1. Otherwise, an auxiliary system generates reparation indexicals by method 3.

Let $c = (W, T)$ a constraint defined by a finite set of tuples. We present here the learning of the bound function $\texttt{minl}_X$, with $X \in W$, the learning of $\texttt{maxl}_X$ being done similarly. An example set E is built such that $\lceil T \rceil \subseteq E \subseteq Int_W$. The user chooses then one function family to express the bound function $\texttt{minl}_X$. A genetic algorithm is used to find the coefficients of this function in order to minimize $\sum_{s \in E} minbc_c^X(s) - \texttt{minl}_X(s)$. A genetic algorithm is a loop composed of a sequence of three phases: selection, cross-over and mutation. In our implementation, the algorithm stops when no modification occurs during a certain lap of time or can be manually stopped by the user.

Inside the genetic algorithm, a population of strings is used to represent a set of candidate operators. Each string is composed of the sequence of binary digits of the coefficients. In figure 3 is depicted a string for the three coefficients of a linear operator. The domain of each coefficient is carefully chosen to ensure the monotonicity of the function.

```
01101001 | 11100100 | 00011101
    a          b          c
```

Fig. 3. A string

Our genetic algorithm maximizes a criterion called *quality*. In order to minimize the distance, we define the quality as the inverse of the distance:

$$Q(\min l_X) = \frac{1}{1 + d(minbc_c^X, \mathtt{minl}_X)}$$

An operator is incorrect if there exists a singletonic state where the expressed function reduces more than bound-consistency, or in other terms for the min bound, if $\exists s \in \lceil T \rceil$ such that $\mathtt{minl}_X(s) > minbc_c^X(s)$. Incorrect operators are given a quality of -1 and eliminated.

Selection is classically obtained by giving each string a chance to reproduce proportional to its relative quality inside the population. Cross-over for two strings containing n coefficients is done by choosing $m \leq n$ bits such that no more than one bit is chosen in a coefficient. Each chosen bit is then exchanged between the two strings. In addition, two mutation operators are used: n-points mutation which randomly changes n bits (one for each coefficient) in one string and continuous mutation which increments or decrements a coefficient by one. New generations replace old ones.

The learning time for all bounds vary between 2 mn and 2 h on a Pentium 4, 2GHz, 512 MB, according to the arity of the constraint and the form of the bounds. Our implementation directly generates the `.fd` files to be linked with the Prolog part of the program.

4 Examples and Discussion

4.1 Examples

In this section are some examples of learned indexicals. All tests have been done on a Pentium 4, 2 GHz, 512MB running Linux. Our tests consists in trying all possible reductions of boxes included in a n-dimensional parallelepiped, n being the arity of the constraint.

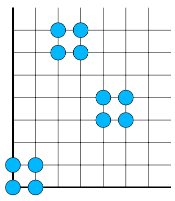


Fig. 4. A disjunctive constraint

Example 21 (Linear constraints). The first example deals with a user defined global constraint defined by the following conjunction: $X - Y > 3, X - Y < 30, 3 * X - Y > 50, 5 * Y - X < 120, X < 45, Y > 5$. When treated globally as a two dimensional polyhedra, these constraints allow a better pruning than the above decomposition. On the domain $[0..49] \times [0..49]$, our system generates the following operators:

```
X in 6*min(Y)/<20+18 .. -10/<(1 + max(Y))+45
Y in 10/<(max(X)+1)+6 .. -590/<(max(X)+1)+47
```

Reparation operators are generated by hand from Gnu-Prolog indexicals following method 1 (`min` and `max` are replaced by `val`). Here is the indexical for X:

```
X in val(Y)+4 .. 44 & 0 .. val(Y)+29 &
     50+val(Y)/>3+1 .. 44 & 5*val(Y)-119 .. 44
```

When trying all reductions on boxes included in $[16, 25] \times [5, 26]$, the learned operators ran in 290 ms, while the non-decomposed constraints ran in 400 ms. The speed-up is 1.37.

Example 22 (Cone). The handling of non-linear constraints is non-optimal in Gnu-Prolog, as in most constraint languages. On the constraint $c_{cone} : (X - 5)^2 + (Y - 5)^2 = Z^2/4$ defining a cone, we get the three following indexicals:

```
X in 7*min(Y)/<20-max(Y)/<20-2*max(Z)/<20
     .. 9-11*min(Y)/<20+8*max(Z)/<20
Y in 7*min(X)/<20-2*max(Z)/<20
     .. 12-15*min(X)/<20+max(X)/<20+6*max(Z)/<20
Z in -2+3*min(X)/<20+11*min(Y)/<20 .. 18
```

On this example, running times were too fast for a direct measure, so we measured a sequence of executions. It gives in average : 190 s for the built-in FD solver and 137 s for the learned linear solver. The speed-up is 1.38. The reparation indexical is implemented with method 3 and is generated automaticaly from the set of solutions.

Example 23 (Cubes). This example of disjunctive constraint is represented in figure 4. We generated the following linear operators:

```
X in 0+15*min(Y)/<20-9*max(Y)/<20 .. 6-9*min(Y)/<20+max(Y)/<20
Y in -1+21*min(X)/<20-4*max(X)/<20 .. 7-5*min(X)/<20+2*max(X)/<20
```

The table constraint ran in 1690 ms, while the learned linear solver ran in 650 ms, producing a speed-up of 2.6. On three dimensional cubes, the piecewise linear operators are better than the linear ones. The reparation indexical is also generated following method 3.

Here is the Gnu-Prolog program we use for testing the constraint `cone` of example 22 (other constraints are tested similarly). In this program, `cone` is the learned constraint and is defined by the three reduction indexicals given in example 22. Since we learn a pre-consistency, `cone` is an over-approximation of the original constraint c_{cone} ($sol(c_{cone}) \subset sol(\texttt{cone})$). The constraint `reparation` consists of the reparation indexical implemented with the C conditional operators as described by method 3.

```
go(X,Y,Z):-
  % generating all boxes
  for(Xmin,0,9),
  for(Xmax,Xmin,9),
  for(Ymin,0,9),
  for(Ymax,Ymin,9),
  for(Zmin,0,9),
  for(Zmax,Zmin,9),
  % definition of domain for each variable
  fd_domain(X,Xmin,Xmax),
  fd_domain(Y,Ymin,Ymax),
  fd_domain(Z,Zmin,Zmax),
  % adding the reduction and reparation indexicals
  fd_tell(cone(X,Y,Z)),
  fd_tell(reparation(X,Y,Z,R)),
  R#>0,
  fd_labeling([X,Y,Z]).
```

This program is compared in performance with the same program written in Gnu-Prolog but which uses directly the constraint c_{cone} expressed with the built-in arithmetic constraints according to the formula.

4.2 Related Work

Solver learning has been first introduced by [4] in which they automatically generate a set of rules from the tuples defining a constraint. These rules, called equality and membership rules, have a fixed form and are lower-level than our proposal. The complexity of their generation limits them to small finite domains such as booleans.

The system PropMiner [1, 2] is devoted to the learning of Constraint Handling Rules [14]. The solver produced by PropMiner is often very readable, especially when a small number of rules are produced, since the implication directly express the propagation. While being less general in theory since we only deal with finite domains, our method generates only one rule for each variable, which may be an advantage in some cases.

Generalized Constraint Propagation [17] as implemented in the Propia library of Eclipse uses least common generalization in order to compile a Prolog predicate into propagators using constructive disjuction. Their most powerful analysis is able to infer arc-consistency coded by the `element` constraint using hidden variable encoding (actually, it corresponds to the implementation of the table constraint by Gnu-Prolog).

In contrast with exact algorithms to compute arc-consistency such as the `CN` algorithm, [19], GAC4 [20], or the GAC-scheme [6], our method only computes an approximation of the desired consistency. This is still useful since the trade-off between the pruning made by the consistency and the efforts needed to enforce it are largely dependent of the nature and the size of the problem.

The theoretical framework presented in this paper allows to express also the approximation of arc-consistency. In this case, we have proposed an approximation using clusters [16] and another one based on rule-based programming [18]. Both methods propose more dramatic speed-ups than approximation of bound-consistency on their dedicated constraints. In contrast, [9] propose another method based on a specific clustering but only as an alternate representation for exact arc-consistency, also like [5]. Moreover, ad-hoc approximate consistencies have also been considered for specific hard-to-propagate constraints, like [22] for the knapsack constraint or bound-consistency for the `alldifferent` constraint [21].

4.3 Discussion

Besides the above example, there is some cases where the process does not work well. First, and as expected, dense constraints are not well reduced by bound-consistency. Since we compute an approximation of it, our operators are even less efficient. An example is the well-known `alldifferent` constraint.

Another problem comes from large tables for which we generate large reparation functions. Since we have a quasi-native C implementation, the performances are quite good but it could be interesting to implement an index and a dichotomic search in order to mimic Prolog's treatment of facts. An other solution would be to work with the complement and to reject the tuples which are accepted by the pre-consistency and should normally be rejected by the constraint.

An other problem comes from a relative weakness of expressivity of the fixed forms we chose for the operators. Yet the instantiation we propose here can be extended to handle more powerful operators, like guarded indexicals coupled with a monotonicity analysis, or even using very different representation like decision trees or neural networks. In such an extended framework, expressions can be found for intermediate consistencies between bound and arc-consistency and which still have a reasonable complexity of evaluation. Our investigation of this topic shows that interesting speed-ups can be achieved [16, 18].

The proposed method does not cope well with the approximation of consistencies stronger than arc-consistency. Actually, arc-consistency is the strongest consistency which can be computed using a single constraint at a time. Hence the only way to build more powerful operators would be to consider the conjunction of two or more constraints at a time as if it was a global constraint and to approximate bound- or arc-consistency for the resulting constraint. It may yield the same propagation than strong consistency but does not reflect the spirit of the mechanism. However, this limitation is also the one of the indexicals, or more generally of the "constraint as operator" formalism used in most solvers. But in the more general rewriting formalism of CHRs [14], the PropMiner system [1] can find stronger propagators. For example, for the boolean constraint `and`, it can generate the following rule:

```
and(X,Y,Z), X ≠ Y  ==>  Z = 0
```

Rules in CHRs usually communicate by adding or removing atoms in or from the store, but this rule can be fired if `and(X,Y,Z)` and `X ≠ Y` are originally in the store, making a deduction impossible to indexicals.

At last, it may be interesting to know when learning operators could be an issue. Table constraints come with their associated arc-consistency operators (called `fd_relation` in Gnu-Prolog). But it is known that a good overall performance is obtained by a carefull balance between the reduction power of the operators and their computational complexity. Pruning values is important but sometimes, an inconsistent value is so hard to find that it is better to keep the work for the search mechanism. This is the reason why bound-consistency is used. But this trade-off is difficult to find since search and consistency are strongly mutually dependent. The form of the bounds used in the learning framework must be meaningful with respect to the target constraint and consistency. In order to find a suitable form, a study of the constraint is necessary, but is somehow tricky since there is no simple link between the form of the bound function and the constraint (see example 13). There is no "best" consistency for all CSPs. Depending on the problem instance and size, on the search strategy, on the complexity to enforce a desired consistency, it is easy to find examples in which a consistency performs better than another one, even if it achieves a weaker pruning. Our present work (and also for arc-consistency [16, 18]) is devoted to provide automatically constructed alternatives for bound- and arc-consistency which can be useful on some problems — or not, depending on the problem type and of the instance. For all these reasons, we consider this method as an environment tool for Constraint Programming. Our examples demonstrate that the technique can be useful in some cases, but their success is difficult to foresee, as for many search strategies.

5 Conclusion

We have presented a general, language-independent framework for finite domains solver learning and an instantiation to the learning of bound consistency with

Gnu-Prolog indexicals. The results are encouraging for this type of techniques since there exists little work aiming at a better propagation of user-defined global constraints. We also believe that this kind of techniques could be part of a constraint modeling environment. The tuning part of a model could then include the identification of sensitive parts which could be locally improved with minimal efforts.

The Solar system extends our previous work called GA-ILearner [10]. We propose here two main improvements, besides a more general theoretical framework: the possibility of using only a sample of the example space while still ensuring the correctness of the learned solver and the reparation method. It follows that our new system is able to handle a much larger example space (larger constraint arity and larger domains) and nevertheless yields a better solver.

Acknowledgements

The authors want to thank:

- Ercim's anonymous referees for pointing out a mistake and for their comments which allow to greatly improve the readability of the paper.
- Daniel Diaz for his kind explanations on Gnu-Prolog's internals and undocumented features and for suggesting the second reparation method.
- Lionel Martin, Gérard Ferrand and Michel Bergère for fruitful discussions.
- French CNRS for its grant ATIP 2JE095.

References

1. Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based solvers for intensionally defined constraints. *International Journal on Artificial Intelligence Tools*, 11(2):283–302, 2002.
2. Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based constraint solvers over finite domains. *Transaction on Computational Logic*, 2003. accepted for publication.
3. K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
4. K.R. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In Joxan Jaffar, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 58–72, Alexandria, Virginia, USA, 1999. Springer.
5. Roman Barták and Roman Mecl. Implementing propagators for tabular constraints. In Francesca Rossi Péter Szeredi Krzysztof R. Apt, Francois Fages and József Váncza, editors, *Joint Annual Workshop of the ERCIM Working Group on Constraints and the CoLogNET area on Constraint and Logic Programming*, pages 69–83, MTA SZTAKI, Budapest, Hungary, 2003.
6. Christian Bessière and Jean-Charles Régin. Arc-consistency for general constraint networks: preliminary results. In *International Joint Conference on Artificial Intelligence*, pages 398–404, Nagoya, Japan, 1997. Morgan Kaufmann.

7. Björn Carlson, Mats Carlsson, and Daniel Diaz. Entailment of finite domain constraints. In Pascal Van Hentenryck, editor, *International Conference on Logic Programming*, pages 339–353, Santa Margherita Ligure, Italy, 1994. MIT Press.
8. Mats Carlsson, Greger Ottosson, and Björ Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 191–206, Southampton, UK, September 3-5 1997. Springer.
9. Kenil C.K. Cheng, Jimmy H.M Lee, and Peter J. Stuckey. Box constraint collections for adhoc constraints. In Francesca Rossi, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *LNCS*, pages 214–228, Kinsale, County Cork, IE, Sept/Oct 2003. Springer.
10. Thi-Bich-Hanh Dao, Arnaud Lallouet, Andrei Legtchenko, and Lionel Martin. Indexical-based solver learning. In Pascal van Hentenryck, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, pages 541–555, Ithaca, NY, USA, Sept. 7 - 13 2002. Springer.
11. Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.
12. Daniel Diaz. Personal communication, 2003.
13. Daniel Diaz and Philippe Codognet. Design and implementation of the Gnu-Prolog system. *Journal of Functional and Logic Programming*, 2001(6), 2001.
14. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
15. Arnaud Lallouet, Thi-Bich-Hanh Dao, Andreï Legtchenko, and AbdelAli Ed-Dbali. Finite domain constraint solver learning. In Georg Gottlob, editor, *International Joint Conference on Artificial Intelligence*, pages 1379–1380, Acapulco, Mexico, 2003. AAAI Press. Poster.
16. Arnaud Lallouet, Andreï Legtchenko, Thi-Bich-Hanh Dao, and AbdelAli Ed-Dbali. Intermediate (learned) consistencies. In Francesca Rossi, editor, *International Conference on Principles and Practice of Constraint Programming*, number 2833 in LNCS, pages 889–893, Kinsale, County Cork, Ireland, 2003. Springer. Poster.
17. Thierry Le Provost and Mark Wallace. Generalized constraint propagation over the CLP Scheme. *Journal of Logic Programming*, 16:319–359, 1993.
18. Andreï Legtchenko. Delaying big operators in order to construct some new consistencies. In Thom Früwirth Slim Abdennadher and Arnaud Lallouet, editors, *International Workshop on Rule-Based Constraint Reasoning and Programming*, pages 98–107, Kinsale, County Cork, Ireland, 2003.
19. Alan K. Mackworth. On reading sketch maps. In R. Reddy, editor, *International Joint Conference on Artificial Intelligence*, pages 598–606, Cambridge, MA, USA, 1977. William Kaufmann.
20. Roger Mohr and Gérald Masini. Good old discrete relaxation. In Yves Kodratoff, editor, *European Conference on Artificial Intelligence*, pages 651–656, Munich, Germany, 1988. Pitmann Publishing.
21. Jean-Francois Puget. A fast algorithm for the bound consistency of alldiff constraints. In *National Conference on Artificial Intelligence*, pages 359–366, Madison, Wisconsin, USA, July 26-30 1998. AAAI Press.
22. Meinolf Sellmann. Approximated consistency for knapsack constraints. In Francesca Rossi, editor, *International Conference on Principles and Practice of Constraint Programming*, number 2833 in LNCS, pages 679–693, Kinsale, County Cork, Ireland, 2003. Springer.
23. P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). draft, 1991.

Abstracting Soft Constraints: Some Experimental Results on Fuzzy CSPs

Stefano Bistarelli[1,2], Francesca Rossi[3], and Isabella Pilan[3]

[1] Istituto di Informatica e Telematica, CNR, Pisa, Italy
Stefano.Bistarelli@iit.cnr.it
[2] Dipartimento di Scienze
Universitá degli Studi "G. D'annunzio" di Chieti-Pescara, Italy
bista@sci.unich.it
[3] Dipartimento di Matematica Pura ed Applicata
Università di Padova, Italy
{frossi,ipilan}@math.unipd.it

Abstract. Soft constraints are very flexible and expressive. However, they may also be very complex to handle. For this reason, it may be convenient in several cases to pass to an abstract version of a given soft problem, and then bring some useful information from the abstract problem to the concrete one. This will hopefully make the search for a solution, or for an optimal solution, of the concrete problem, faster.
In this paper we review the main concepts and properties of our abstraction framework for soft constraints, and we show some experimental results of its application to the solution of fuzzy constraints.

1 Introduction

Soft constraints allow to model faithfully many real-life problems, especially those which possess features like preferences, uncertainties, costs, levels of importance, and absence of solutions. Formally, a soft constraint problem (SCSP) is just like a classical constraint problem (CSP), except that each assignment of values to variables in the constraints is associated to an element taken from a set (usually ordered). These elements will then directly represent the desired features, since they can be interpreted, for example, as levels of preference, or costs, or levels of certainty.

SCSPs are more expressive than classical CSPs, but they are also more difficult to process and to solve, mainly because they are optimization rather than satisfaction problems. For this reason, it may be convenient to work on a simplified version of the given problem, trying however to not loose too much information. In [3, 1, 2], an abstraction framework for soft constraints has been proposed, where, from a given SCSP, a new simpler SCSP is generated representing an "abstraction" of the given one. Then, the abstracted version is processed (that is, solved or checked for consistency), and some information gathered during the solving process is brought back to the original problem, in order to transform it into a new (equivalent) problem easier to solve.

All this process has the main aim of finding an optimal solution, or an approximation of it, for the original SCSP, with less time or space w.r.t. a solution process that does not involve the abstraction phase. It is also useful when we don't have any solver for the class of soft problems we are interested in, but we do have it for another class, to which we can abstract to. In this way, we rely on existing solvers to automatically build new solvers.

Many properties of the abstraction framework have been proven in [3]. The most interesting one, which will be used in this paper, is that, given any optimal solution of the abstract problem, we can find upper and lower bounds for an optimal solution for the concrete problem. If we are satisfied with these bounds, we could just take the optimal solution of the abstract problem as a reasonable approximation of an optimal solution for the concrete problem.

In this paper we extend this and other results described in [3] to build a new algorithm to solve the concrete problem by using abstraction steps. More in detail, we prove that using the value of the optimal tuple in the abstract problem as a bound for the abstraction mapping, leads to building each time new abstract problem with better and better corresponding optimal concrete solution. When no more solutions are found, we can be sure that the last found optimum in the abstract problem is indeed the optimal solution of the concrete one.

Besides devising a new solving algorithm based on such results and on the abstraction framework, we also describe the results of several experiments which show the behavior of three variants of such algorithm over fuzzy constraint problems [7, 13, 14]. In these experiments, fuzzy problems are abstracted into classical CSPs, and solved via iterative abstraction of the given problem in different classical CSPs. The behavior of the three versions of our algorithm is then compared to that of Conflex, a solver for fuzzy CSPs. This paper is an extended and improved version of [12].

2 Soft Constraints

A soft constraint [5] is just a classical constraint where each instantiation of its variables has an associated value from a partially ordered set. Combining constraints will then have to take into account such additional values, and thus the formalism has also to provide suitable operations for combination ($\times$) and comparison (+) of tuples of values and constraints. This is why this formalization is based on the concept of semiring, which is just a set plus two operations satisfying certain properties: $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$.

If we consider the relation $\leq_S$ over A defined as $a \leq_S b$ iff $a + b = b$, then we have that:

- $\leq_S$ is a partial order;
- $+$ and $\times$ are monotone on $\leq_S$;
- $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum;
- $\langle A, \leq_S \rangle$ is a complete lattice and $+$ is its lub.

Moreover, if $\times$ is idempotent, then $\langle A, \leq_S \rangle$ is a complete distributive lattice and $\times$ is its glb. Informally, the relation $\leq_S$ gives us a way to compare (some of the) values in the set A. In fact, when we have $a \leq_S b$, we will say that *b is better than a*. Extending the partial order $\leq_S$ among tuples of values, also a partial order among constraints is induced.

Given a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a finite set D (the domain of the variables), and an ordered set of variables V, a constraint is a pair $\langle def, con \rangle$ where $con \subseteq V$ and

$def : D^{|con|} \rightarrow A$. Therefore, a constraint specifies a set of variables (the ones in *con*), and assigns to each tuple of values of D of these variables an element of the semiring set A. This element can then be interpreted in several ways: as a level of preference, or as a cost, or as a probability, etc. The correct way to interpret such elements depends on the choice of the semiring operations.

Constraints can be compared by looking at the semiring values associated to the same tuples: Consider two constraints $c_1 = \langle def_1, con \rangle$ and $c_2 = \langle def_2, con \rangle$, with $|con| = k$. Then $c_1 \sqsubseteq_S c_2$ if for all k-tuples t, $def_1(t) \leq_S def_2(t)$. The relation $\sqsubseteq_S$ induces a partial order. In the following we will also use the obvious extension of this relation to sets of constraints, and also to problems (seen as sets of constraints).

Note that a classical CSP is a SCSP where the chosen c-semiring is:

$$S_{CSP} = \langle \{false, true\}, \vee, \wedge, false, true \rangle.$$

Fuzzy CSPs [7, 13, 14], which will be the main subject of this paper, can instead be modeled in the SCSP framework by choosing the c-semiring: $S_{FCSP} = \langle [0,1], max, min, 0, 1 \rangle$.

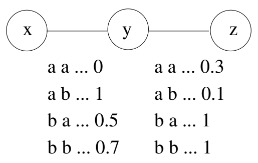
Given two constraints $c_1 = \langle def_1, con_1 \rangle$ and $c_2 = \langle def_2, con_2 \rangle$, their *combination* $c_1 \otimes c_2$ is the constraint $\langle def, con \rangle$ defined by $con = con_1 \cup con_2$ and $def(t) = def_1(t \downarrow^{con}_{con_1}) \times def_2(t \downarrow^{con}_{con_2})$. In words, combining two constraints means building a new constraint involving all the variables of the original ones, and which associates to each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate sub-tuples.

Given a constraint $c = \langle def, con \rangle$ and a subset I of V, the *projection* of c over I, written $c \Downarrow_I$, is the constraint $\langle def', con' \rangle$ where $con' = con \cap I$ and $def'(t') = \sum_{t/t \downarrow^{con}_{I \cap con} = t'} def(t)$. Informally, projecting means eliminating some variables. This is done by associating to each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables.

The *solution* of a SCSP problem $P = \langle C, con \rangle$ is the constraint $Sol(P) = (\bigotimes C) \Downarrow_{con}$: we combine all constraints, and then project over the variables in *con*. In this way we get the constraint over *con* which is "induced" by the entire SCSP. Optimal solutions are those solutions which have the best semiring element among those associated to solutions. The set of optimal solutions of an SCSP P will be written as $Opt(P)$. In the following, we will sometimes call "a solution" one tuple of domain values for all the problem's variables (over *con*), plus its associated semiring element. Figure 1 shows an example of fuzzy CSP and its solutions.

Consider two problems P_1 and P_2. Then $P_1 \sqsubseteq_P P_2$ if $Sol(P_1) \sqsubseteq_S Sol(P_2)$. If $P_1 \sqsubseteq_P P_2$ and $P_2 \sqsubseteq_P P_1$, then they have the same solution, thus we say that they are equivalent and we write $P_1 \equiv P_2$.

SCSP problems can be solved by extending and adapting the technique usually used for classical CSPs. For example, to find the best solution we could employ a branch-and-bound search algorithm (instead of the classical backtracking), and also the successfully used propagation techniques, like arc-consistency [11], can be generalized to be used for SCSPs. The detailed formal definition of propagation algorithms (sometimes called also *local consistency* algorithms) for SCSPs can be found in [5]. For the purpose of this



Solutions:

a a a ... min(0,0.3) = 0
a a b ... min(0,0.1) = 0
a b a ... min(1,1) = 1
a b b ... min(1,1) = 1
b a a ... min(0.5,0.3) = 0.3
b a b ... min(0.5,0.1) = 0.1
b b a ... min(0.7,1) = 0.7
b b b ... min(0.7,1) = 0.7

Best solutions:
a b a ... 1
a b b ... 1

Fig. 1. A fuzzy CSP and its solutions.

paper, what is important to say is that a *propagation rule* is a function which, taken an SCSP, solves a subproblem of it. It is possible to show that propagation rules are idempotent, monotone, and intensive functions (over the partial order of problems) which do not change the solution set. Given a set of propagation rules, a local consistency algorithm consists of applying them in any order until stability. It is possible to prove that local consistency algorithms defined in this way have the following properties if the multiplicative operation of the semiring is idempotent: equivalence, termination, and uniqueness of the result.

Thus we can notice that the generalization of local consistency from classical CSPs to SCSPs concerns the fact that, instead of deleting values or tuples, obtaining local consistency in SCSPs means changing the semiring values associated to some tuples or domain elements. The change always brings these values towards the worst value of the semiring, that is, the **0**. Thus, it is obvious that, given an SCSP problem P and the problem P' obtained by applying some local consistency algorithm to P, we have $P' \sqsubseteq_S P$.

3 Abstraction

The main idea [6] is to relate the concrete and the abstract scenarios by a pair of functions, the *abstraction* function α and the *concretization* function γ, which form a Galois connection.

Let $(\mathcal{C},\sqsubseteq)$ and $(\mathcal{A},\leq)$ be two posets (the concrete and the abstract domain). A Galois connection $\langle\alpha,\gamma\rangle : (\mathcal{C},\sqsubseteq) \rightleftharpoons (A,\leq)$ is a pair of maps $\alpha : \mathcal{C} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathcal{C}$ such that 1. α and γ are monotonic, 2. for each $x \in \mathcal{C}, x \sqsubseteq \gamma(\alpha(x))$ and 3. for each $y \in \mathcal{A}, \alpha(\gamma(y)) \leq y$. Moreover, a Galois insertion (of $\mathcal{A}$ in $\mathcal{C}$) $\langle\alpha,\gamma\rangle : (\mathcal{C},\sqsubseteq) \rightleftharpoons (\mathcal{A},\leq)$ is a Galois connection where $\alpha\cdot\gamma$ is the identity over $\mathcal{A}$, that is, $Id_{\mathcal{A}}$.

An example of a Galois insertion can be seen in Figure 2. Here, the concrete lattice is $\langle[0,1],\leq\rangle$, and the abstract one is $\langle\{0,1\},\leq\rangle$. Function α maps all real numbers in $[0,0.5]$ into 0, and all other integers (in $(0.5,1]$) into 1. Function γ maps 0 into 0.5 and 1 into 1.

Consider a Galois insertion from $(\mathcal{C},\sqsubseteq)$ to $(\mathcal{A},\leq)$. Then, if $\sqsubseteq$ is a total order, so is $\leq$.

Most of the times it is useful, and required, that the abstract operators show a certain relationship with the corresponding concrete ones. This relationship is called *local cor-*

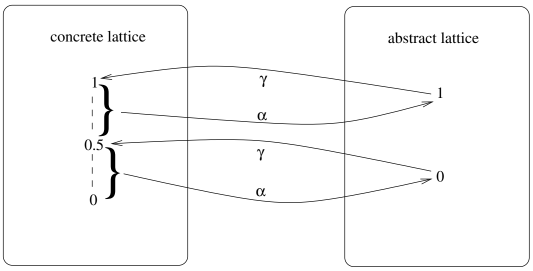


Fig. 2. A Galois insertion.

rectness. Let $f : \mathcal{C}^n \to \mathcal{C}$ be an operator over the concrete lattice, and assume that $\tilde{f}$ is its abstract counterpart. Then $\tilde{f}$ is locally correct w.r.t. f if $\forall x_1, \ldots, x_n \in \mathcal{C}, f(x_1, \ldots, x_n) \sqsubseteq \gamma(\tilde{f}(\alpha(x_1), \ldots, \alpha(x_n)))$.

3.1 Abstracting Soft CSPs

The main idea of the abstraction framework presented in [3] is very simple: we just want to pass, via the abstraction, from an SCSP P over a certain semiring S to another SCSP $\tilde{P}$ over the semiring $\tilde{S}$, where the lattices associated to $\tilde{S}$ and S are related by a Galois insertion as shown above.

Consider the *concrete* SCSP problem $P = \langle C, con \rangle$ over semiring S, where

- $S = \langle A, +, \times, 0, 1 \rangle$ and
- $C = \{c_0, \ldots, c_n\}$ with $c_i = \langle con_i, def_i \rangle$ and $def_i : D^{|con_i|} \to A$;

we define an *abstract* SCSP problem $\tilde{P} = \langle \tilde{C}, con \rangle$ over the semiring $\tilde{S}$, where

- $\tilde{S} = \langle \tilde{A}, \tilde{+}, \tilde{\times}, \tilde{0}, \tilde{1} \rangle$;
- $\tilde{C} = \{\tilde{c_0}, \ldots, \tilde{c_n}\}$ with $\tilde{c_i} = \langle con_i, \tilde{def_i} \rangle$ and $\tilde{def_i} : D^{|con_i|} \to \tilde{A}$;
- if $L = \langle A, \leq \rangle$ is the lattice associated to S and $\tilde{L} = \langle \tilde{A}, \tilde{\leq} \rangle$ the lattice associated to $\tilde{S}$, then there is a Galois insertion $\langle \alpha, \gamma \rangle$ such that $\alpha : L \to \tilde{L}$;
- $\tilde{\times}$ is locally correct with respect to $\times$.

Notice that the kind of abstraction we consider in this paper does not change the structure of the SCSP problem. The only thing that is changed is the semiring.

Notice also that, given two problems over two different semirings, there may exist zero, one, or also many abstractions (that is, a Galois insertion between the two semirings) between them. This means that given a concrete problem over S and an abstract semiring $\bar{S}$, there may be several ways to abstract such a problem over $\bar{S}$.

Example 1. As an example, consider any SCSP over the semiring for optimization $\langle \mathcal{R}^- \cup \{-\infty\}, \max, +, -\infty, 0 \rangle$ and suppose we want to abstract it onto the semiring for fuzzy reasoning $\langle [0,1], max, min, 0, 1 \rangle$. In other words, instead of computing the maximum of the sum of all costs (which are negative reals), we just want to compute the maximum of their minimum value, and we want to normalize the costs over $[0,1]$. Notice that the abstract problem has an idempotent $\times$ operator (which is the min). This means that in the abstract framework we can perform local consistency over the problem in order to find inconsistencies.

Example 2. Another example is the abstraction from the fuzzy semiring to the classical one, which will be widely used in the rest of this paper:

$$S_{CSP} = \langle \{0,1\}, \vee, \wedge, 0, 1 \rangle.$$

Here function α maps each element of $[0,1]$ into either 0 or 1. For example, one could map all the elements in $[0,x]$ onto 0, and all those in $(x,1]$ onto 1, for some fixed x. Figure 2 represents this example with $x = 0.5$.

3.2 Properties of the Abstraction

We will now summarize the main results about the relationship between a concrete problem and an abstraction of it.

Let us consider the scheme depicted in Figure 3. Here and in the following pictures, the left box contains the lattice of concrete problems, and the right one the lattice of abstract problems. The partial order in each of these lattices is shown via dashed lines. Connections between the two lattices, via the abstraction and concretization functions, is shown via directed arrows. In the following, we will call $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ the concrete semiring and $\tilde{S} = \langle \tilde{A}, \tilde{+}, \tilde{\times}, \tilde{\mathbf{0}}, \tilde{\mathbf{1}} \rangle$ the abstract one. Thus we will always consider a Galois insertion $\langle \alpha, \gamma \rangle : \langle A, \leq_S \rangle \rightleftharpoons \langle \tilde{A}, \leq_{\tilde{S}} \rangle$.

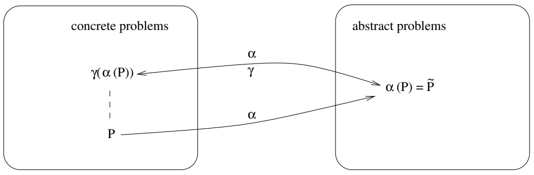


Fig. 3. The concrete and the abstract problem.

In Figure 3, P is the starting SCSP problem. Then with the mapping α we get $\tilde{P} = \alpha(P)$, which is an abstraction of P. By applying the mapping γ to $\tilde{P}$, we get the problem $\gamma(\alpha(P))$. Let us first notice that these two problems (P and $\gamma(\alpha(P))$) are related by a precise property:

$$P \sqsubseteq_S \gamma(\alpha(P)).$$

Notice that this implies that, if a tuple in $\gamma(\alpha(P))$ has semiring value $\mathbf{0}$, then it must have value $\mathbf{0}$ also in P. This holds also for the solutions, whose semiring value is obtained by combining the semiring values of several tuples. Therefore, by passing from P to $\gamma(\alpha(P))$, no new inconsistencies are introduced. However, it is possible that some inconsistencies are forgotten.

Example 3. Consider the abstraction from the fuzzy to the classical semiring, as described in Figure 2. Then, if we call P the fuzzy problem in Figure 1, Figure 4 shows the concrete problem P, the abstract problem $\alpha(P)$, and its concretization $\gamma(\alpha(P))$. It is easy too see that, for each tuple in each constraint, the associated semiring value in P is lower than or equal to that in $\gamma(\alpha(P))$.

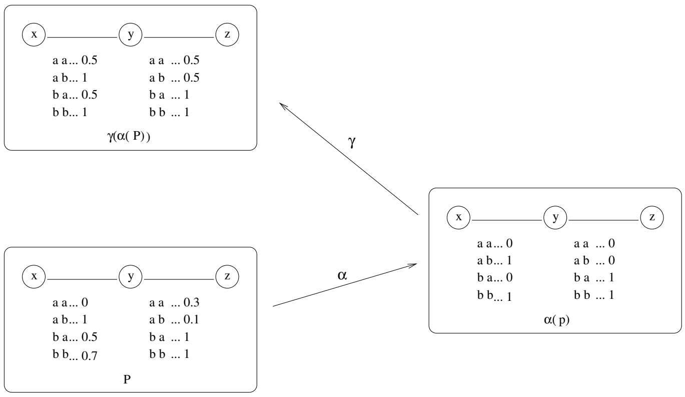


Fig. 4. An example of the abstraction fuzzy-classical.

If the abstraction preserves the semiring ordering (that is, applying the abstraction function and then combining gives elements which are in the same ordering as the elements obtained by combining only), then the abstraction is called *order-preserving*, and in this case there is also an interesting relationship between the set of optimal solutions of P and that of $\alpha(P)$. In fact, if a certain tuple is optimal in P, then this same tuple is also optimal in $\alpha(P)$.

Example 4. Consider again the previous example. The optimal solutions in P are the tuples $\langle a,b,a\rangle$ and $\langle a,b,b\rangle$. It is easy to see that these tuples are also optimal in $\alpha(P)$. In fact, this is a classical constraint problem where the solutions are tuples $\langle a,b,a\rangle$, $\langle a,b,b\rangle$, $\langle b,b,a\rangle$, and $\langle b,b,b\rangle$.

Thus, if we want to find an optimal solution of the concrete problem, we could find all the optimal solutions of the abstract problem, and then use them on the concrete side to find an optimal solution for the concrete problem. Assuming that working on the abstract side is easier than on the concrete side, this method could help us find an optimal solution of the concrete problem by looking at just a subset of tuples in the concrete problem.

Another important property, which holds for any abstraction, concerns computing bounds that approximate an optimal solution of a concrete problem. In fact, any optimal solution, say t, of the abstract problem, say with value $\tilde{v}$, can be used to obtain both an upper and a lower bound of an optimum in P. In fact, we can prove that there is an optimal solution in P with value between $\gamma(\tilde{v})$ and the value of t in P [3, Theorem 29].

Thus, if we think that approximating the optimal value with a value within these two bounds is satisfactory, we can take t as an approximation of an optimal solution of P. Notice that this theorem does not need the order-preserving property in the abstraction, thus any abstraction can exploit this result.

Example 5. Consider again the previous example. Now take any optimal solution of $\alpha(P)$, for example tuple $\langle b,b,b\rangle$. Then the above result states that there exists an optimal solution of P with semiring value v between the value of this tuple in P, which is 0.7, and $\gamma(1) = 1$. In fact, there are optimal solutions with value 1 in P.

However, a better lower bound can be computed in the special case of an abstraction where the semirings are totally ordered and have idempotent multiplicative operations. In this case, any abstraction is order-preserving. In fact, consider an abstraction between totally ordered semirings with idempotent multiplicative operations. Given an SCSP problem P over S, consider an optimal solution of $\alpha(P)$, say t, with semiring value $\tilde{v}$ in $\alpha(P)$. Consider also the set $V = \{v_i \mid \alpha(v_i) = \tilde{v}\}$. Then there exists an optimal solution $\bar{t}$ of P, say with value $\bar{v}$, such that $min(V) \leq \bar{v} \leq max(V)$.

3.3 New Properties

When dealing with the mapping from fuzzy to classical CSPs we can also prove other important results. Consider an abstraction that maps all the semiring values better than the fuzzy value α into 1 (that is, the boolean *true*) and all the fuzzy values worse than or equal than α to 0 (that is, the boolean value *false*). Let us also call P the fuzzy CSP and $\alpha(P)$ the corresponding abstracted CSP. Then we can prove that:

Given an SCSP problem P over the fuzzy semiring, and the corresponding abstract problem $\alpha(P)$ over the boolean semiring, obtained by mapping all values lower than or equal than α to *false* and all the values bigger than α to *true*.

- if $\alpha(P)$ has no solution, problem P has an optimal solution with an associated semiring fuzzy value worse than or equal than α;
- if P has a solution tuple t with associated semiring level α, and $\alpha(P)$ has no solution, tuple t is an optimal solution for P.

These properties will be very useful in devising the three versions of the abstraction-based algorithm we will define in the next section.

4 Solving by Iterative Abstraction

The results of the previous section can be the basis for a constraint solving method, or more precisely a family of methods, where abstraction will be used to compute or to approximate the solution of a concrete problem.

Here we will focus on the version of this solving method which applies to fuzzy CSPs, because our experimental results will focus on this class of soft CSPs. The general version of the algorithm is given in [3].

A method to solve a fuzzy CSP is to reduce the fuzzy problem to a sequence of classical (boolean) CSPs to be solved by a classical solver. This method has been for instance recently implemented in the JFSolver [10].

Let us formalize this algorithm within our abstraction framework. We want to abstract a fuzzy CSP $P = \langle C, con \rangle$ into the boolean semiring. Let us consider the abstraction α which maps the values in [0,0.5] to 0 and the values in]0.5,1] to 1, which is depicted in Figure 2. Let us now consider the abstract problem $\tilde{P} = \alpha(P) = \langle \tilde{C}, con \rangle$. There are two possibilities, depending whether $\alpha(P)$ has a solution or not.

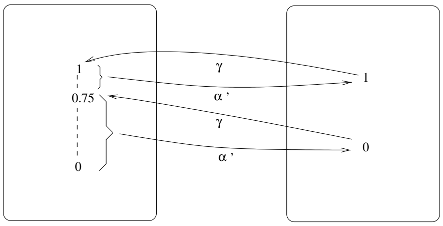


Fig. 5. An abstraction from the fuzzy semiring to the boolean one, cut level = 0.75.

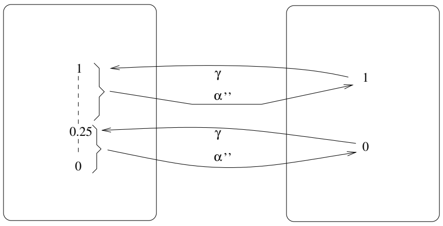


Fig. 6. An abstraction from the fuzzy semiring to the boolean one, cut level = 0.5.

1. If $\alpha(P)$ has a solution, then (by the previous results) P has an optimal solution $\bar{t}$ with value $\bar{v}$, such that $0.5 \leq \bar{v} \leq 1$. We can now further cut this interval in two parts, e.g. [0.5,0.75] and]0.75, 1], and consider now the abstraction α' which maps the values in [0,0.75] to 0 and the values in]0.75,1] to 1, which is depicted in Figure 5. If $\alpha'(P)$ has a solution, then P has a corresponding optimal solution with fuzzy value between 0.75 and 1, otherwise the optimal solution has fuzzy value between 0.5 and 0.75, because we know from the previous iteration that the solution is above 0.5. If tighter bounds are needed, one could further iterate this process until the desired precision is reached.
2. If $\alpha(P)$ has no solution, then (by the previous results) P has an optimal solution $\bar{t}$ with value $\bar{v}$, such that $0 \leq \bar{v} \leq 0.5$. We can now further cut this interval in two parts, e.g. [0,0.25] and]0.25,0.5[, and consider now the abstraction α'' which maps the values in [0,0.25] to 0 and the values in]0.25,1] to 1, which is depicted in Figure 6. If $\alpha''(P)$ has a solution, then P has a corresponding optimal solution with fuzzy value between 0.25 and 0.5, otherwise the optimal solution has fuzzy value between 0 and 0.25. And so on and so forth.

Example 6. For example, consider again the fuzzy constraint problem and the initial abstraction of Figure 4. This abstraction uses a cut level of 0.5 and generates the classical constraint problem in the right part of Figure 4. This problem has solutions (for example, $x = a$, $y = b$, and $z = a$), thus the algorithm above sets the new cut level at 0.75. The new abstracted problem is still soluble: for example, the assignment above is still a solution. Thus the set the new cut level to 0.875. Again, the abstracted problem has solutions (it is actually the same problem as before), so we set the new cut level to 0.9375. The abstracted problem has again solutions. If we have reached the desired precision (assume that we are happy with a tolerance of 0.1, we can conclude the algorithm by reporting the existence of a solution for the initial fuzzy CSPs with value higher than, or equal to 0.9375. More precisely, the iterative algorithm reports that there is a solution in the interval [0.9375,1].

Observe that the dichotomy method used in this example is not the only one that be can used: we can cut each interval not necessarily in the middle but at any point at will (e.g., if some heuristic method allows us to guess in which part of the semiring the optimal solution is). The method will work as well, although the convergence rate, and thus the performance, could be different. In particular we can use the results of the previous section and cut each time at level α corresponding to the value of the tuple t that is optimal in the abstract problem (let's call this heuristic *current best*). We can then continue this procedure until we find no solutions. At this point we are sure that the tuple t found at the previous step is an optimal solution for the concrete problem P.

Example 7. In the example above, this version of the algorithm would work as follows. First, the fuzzy problem is cut at level 0.5, and its abstracted version is the one at the right in Figure 4. This problem has solutions, so we take any solution, for example $x = b$, $y = b$, and $z = b$, and we compute its value in the fuzzy problem. In this case the value is 0.7. By setting the new cut level to 0.7, the new abstracted problem has still solutions (although tuple ¡b,b¿ between x and y has now value 0), so we take any of such solutions, say $x = a$, $y = b$, and $z = b$, and we compute its value in the fuzzy problem, which is 1. Now the new abstracted problem has no solution, so we stop reporting $x = a$, $y = b$, and $z = b$ as an optimal solution for the initial fuzzy problem.

5 Experimental Setting and Results

Our experimental setting involves a generator of fuzzy SCSPs and the implementation of three versions of the iterative algorithm defined in the previous section. More precisely, the three algorithms we will consider are the following ones:

A1: Algorithm A1 sets the cut level α to cut the current interval in two equal parts at each abstraction step, as defined in the previous section. It stops when a precision of 1/10 is reached (that is, the size of the considered interval is smaller than or equal to 0.1).

A2: Algorithm A2 sets the cut level α to the semiring level of the current best solution found. It stops when a precision of 1/10 is reached.

A3: Algorithm A3 behaves as algorithm A2, except that it stops when no solution in the abstract problem is found. At this point, the last solution found is an optimal solution of the concrete problem.

The generator generates binary fuzzy CSPs with a certain number of variables (n), number of values per variable (m), density (d, which is the percentage of constraints over the maximum possible number) and tightness (that is, percentage of tuples with preference 0, denoted by t). For each set of parameters, we generate three instance problems, and we show the mean result on them.

In all our experiments, which have been performed on a Pentium 3 processor at 850 MHz, we solve concrete fuzzy CSPs via Conflex [7] and, within the abstraction-based algorithms, we solve their abstracted boolean versions via Conflex as well. Since Conflex is especially designed to solve fuzzy CSPs, it has some overhead when solving classical CSPs. Thus we may imagine that by using a solver for boolean CSPs we may get a better performance for the proposed algorithms.

Conflex solves fuzzy constraint problems by using a branch and bound approach, combined with constraint propagation. Moreover, it allows users to set a threshold (between 0 and 1) which is used to disregard those solutions with values below the threshold (and thus also to perform pruning in the search tree).

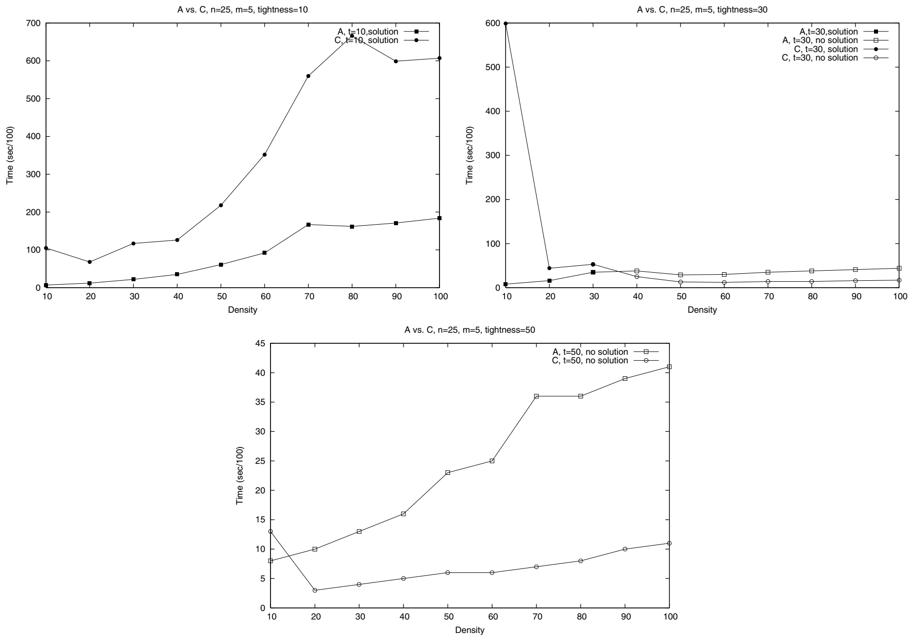


Fig. 7. Time for algorithms A and C, tightness 10%, 30% and 50%.

We start our tests by comparing algorithm A1 to Conflex. Figure 7 shows the time needed to find an optimal solution, or to discover that no solution exists, for both A1 (denoted by A in this pictures since it is the only abstraction-based algorithm) and Conflex (denoted by *C*), with a varying density over the x axis, and varying tightness in the three figures (t=10,30 and 50). The number of variables is fixed to 25, while the domain size is 5. For these experiments, we set the initial threshold 0.01 for algorithm *C* (thus not much pruning is possible initially because of the threshold), and the initial cut level to 0.5. The filled points denote problems with solutions, while the empty points denote problems with no solution.

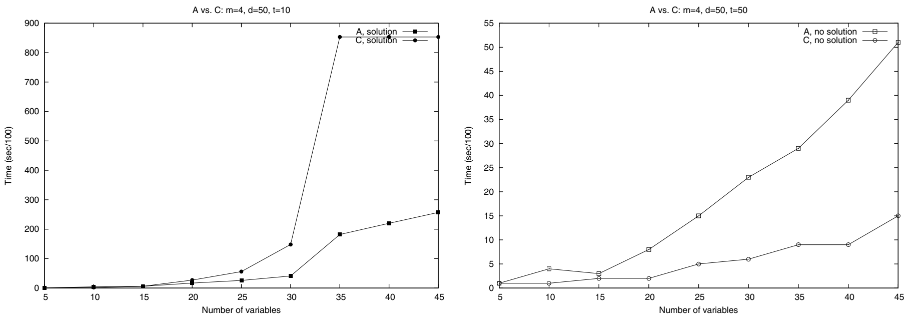


Fig. 8. Time for A and C, varying number of variables, tightness 10% and 50%.

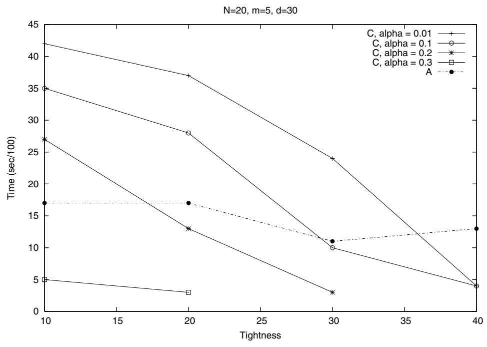


Fig. 9. Time for A and C (with varying threshold), density 30%.

The graphs show very clearly that, in the presence of solutions, method A1 is better, while Conflex is more convenient when there is no solution. This is predictable: in fact, the iterative algorithm A1, in presence of no solution, would nevertheless continue shrinking the interval until the desired precision.

Let us now see what happens when the number of variables varies. The results are shown in Figure 8, where the x axis shows the number of variables. Density is set to 50%, while tightness varies (10% and 50 %). Again, the threshold in C is 0.01 and the initial cut level is 0.5. The graphs show again that the abstraction-based method is convenient in solving problems with solutions, while the classical method (that is, Conflex in our experiments) is better when there is no solution.

One could argue that a threshold of 0.01 given to Conflex is a very bad situation for this algorithm, since it cannot perform almost any pruning because of such a low threshold. However, it has to be noticed that, if we give a threshold which is higher than the value of the optimal solution, then Conflex would not find the optimal solutions. Nevertheless, we run some experiments with different thresholds.

Figure 9 shows the time needed by Conflex with different thresholds (from 0.1 to the first threshold which generates no solution) in different density/tightness scenarios. We can see that algorithm A1 has a balanced performance when the tightness varies,

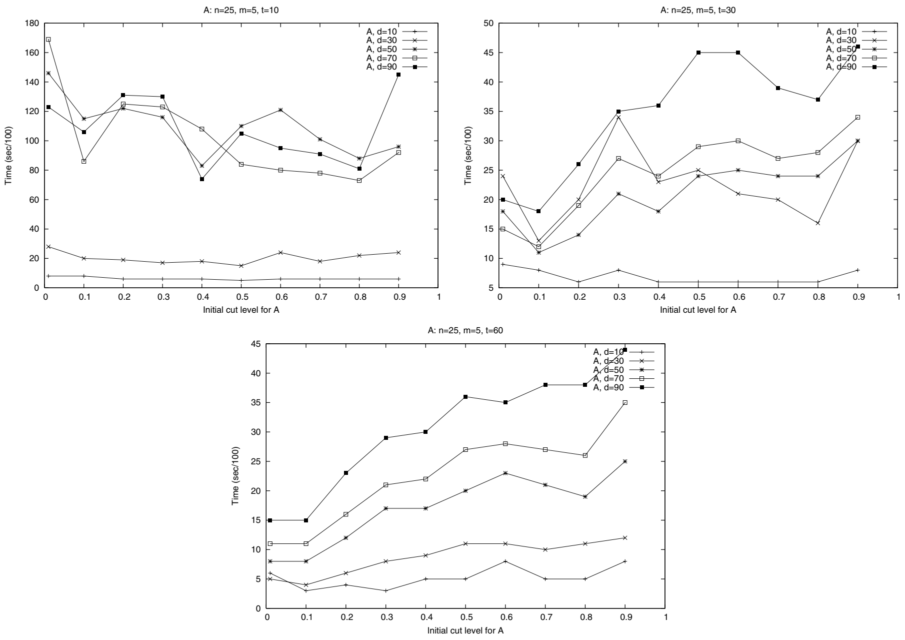


Fig. 10. Time for A, different initial cut levels, tightness 10%, 30%, and 60%.

while Conflex, as predictable, behaves better when the threshold is close to the value of the optimal solution (which can be deduced by the first threshold which generates no solution). Therefore, if we can guess a threshold which is close to the optimal solution value, then using *C* is better. However, in absence of this information, we should rather use the abstraction-based method, which gives a good performance in all situations.

We may also wonder about the influence of the initial cut level, which up to now has always been 0.5, over the behaviour of the iterative abstraction method. Figures 10 shows the time needed for method A1 when tightness, density, and initial cut level vary. It is easy to see that, with high tightness and density, it is better to set a low initial cut level. This is probably because a high tightness and density usually imply low values for the optimal solutions, thus starting with a high cut level would generate more iterations to get down to the interval containing the optimal solution, if any.

We now pass to consider the other two variants of the original abstraction algorithm: A2 and A3. Figure 11 shows the time needed to find an optimal solution, or to discover that no solution exists, for both the iterative algorithms A1, A2, and A3, and C, with a varying density over the x axis, an tightness t = 10%. The number of variables is fixed to 25 and the domain size is 5. For these experiments, we set as before an initial threshold of 0.01 in C and an initial cut level of 0.5. The graphs show how algorithm A1, A2, and A3 have similar performance, and all of them are better than C when the tightness is not too high. With high tightness, algorithm A3 is worse than C. In fact, the iterative algorithm A3 would spend more time in shrinking the intervals until a precise solution is found.

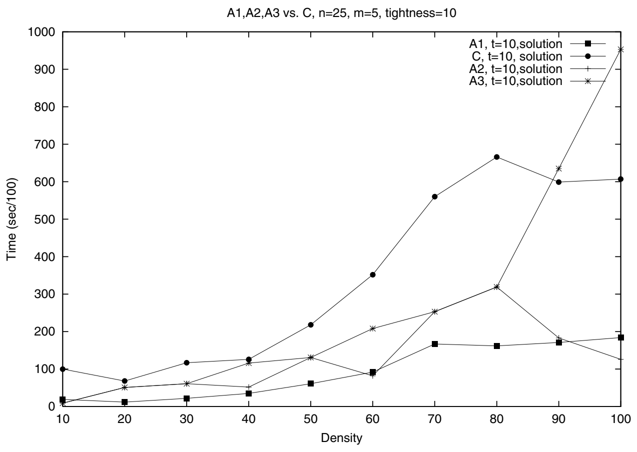


Fig. 11. Time for algorithms A1, A2, A3, and C, tightness 10%.

Let us now compare C to the iterative abstraction algorithms when the number of variables varies (over the x axis). Density is set to 50%, tightness to 10%, and threshold to 0.01. The results are shown in Figure 12. Again, the graphs show that the abstraction-based methods are convenient in solving problems with solutions. We also notice how algorithm A2 is better than A1 when the number of variables increase.

The next experiment shows the comparison of A1, A2, and A3 with C over combinations of densities and tightnesses which generate mostly problems with solutions. Figure 13 shows the results for problems with density 70%. As before, algorithms A1 and A2 are better than C. We recall that algorithm A3 always finds an exact solution, whilst method A1 only stops when the desired precision is reached.

Summarizing, we can learn the following lessons from these last experiments:

– With a small number of variables (25):
  - A2 is more expensive than A1; therefore, it is not worthed to use the value of the abstract solutions to decide the next cut level;
  - A3 is more expensive than A1, but it obtains an optimal solution, not an interval where optimal solutions are contained.
– As the number of variables increases:
  - A2 is less expensive than A1;
  - A3 is more expensive but still convenient w.r.t. C when tightness is not very high.

6 Related Work

Besides Conflex, there exist other systems which allow to solve soft constraints in general and thus also fuzzy constraints.

One is the CLP(FD,S) [9] language, which is a constraint logic programming language where the underlying constraint solver can handle soft constraints. In particular,

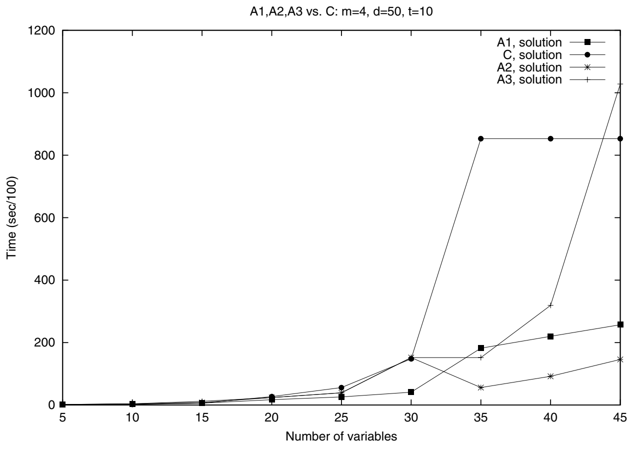


Fig. 12. Time for A1, A2, A3, and C, varying number of variables, tightness 10%.

CLP(FD,S) is a language for modeling and solving semiring-based constraints. The programmer can specify the semiring to be used, thus creating a specific instance of the language, which can handle soft constraints over that semiring. When the semiring to be used has an idempotent combination operator, like for instance for fuzzy CSPs, local propagation techniques are used in the solution process, which is based on a branch and bound approach.

Another one is the Constraint Handling Rules (CHR) [8] system. CHR is a high level formalism to specify constraint solvers and propagation algorithms. It has been originally designed to model and solve classical constraints, but recently [4] it has been extended to handle also semiring-based soft constraints. This extension solves soft constraints by performing local propagation (like node and arc consistency), embedded in one of the two available complete solvers, based on dynamic programming and branch and bound, respectively.

In the experimental work reported in this paper, we did not implement any specific propagation technique or solver: we just solved fuzzy constraint problems by using several times the Conflex solver on classical constraint problems, and we compared this method to using Conflex directly on the given fuzzy problem. For this reason, our results are not directly comparable with [9] or [4].

However, we can try to give an indirect comparison of our results with the CLP(FD,S) system. In fact, in [9] the Conflex system is compared with CLP(FD,S) when solving fuzzy CSPs. The results shown in [9] show that CLP(FD,S) is 3 to 6 times faster than the Conflex system. Our implementation, which is very naive, performs 3 times better than Conflex in average. Thus, we are comparable to CLP(FD,S), which is an optimized system for soft constraints. We plan to implement local propagation techniques during the abstraction steps. We believe that this will burst the performance of our technique, and make it more convenient than CLP(FD,S). We notice however that CLP(FD,S) is unfortunately not maintained any longer, thus it will be difficult to make a fair and complete comparison.

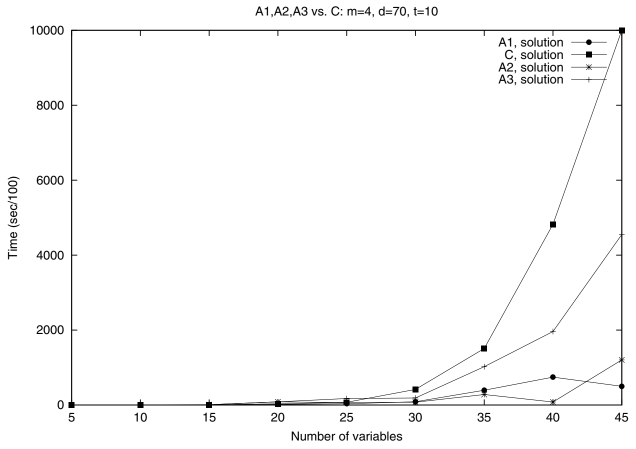


Fig. 13. Time for A1, A2, A3, and C, density 70%, tightness 10%.

7 Conclusions and Future Work

We have run several experiments to study the behavior of three versions of an iterative abstraction-based algorithm to solve fuzzy CSPs. The main lesson learnt from these experiments is that, when we work with problems for which we can guess the existence of some solutions, the abstraction methods are more convenient. This holds also when we don't have any information on the value of the optimal solutions. Among the three versions of the algorithm, the first two (A1 and A2) are the best ones. However, since A2 always finds a solution and not an approximation of it, it is to be chosen.

The iterative abstraction methodology we have tested looks promising and suitable to solve fuzzy CSPs. We recall that our experiments used Conflex for solving both the concrete fuzzy problem instances and also the abstract boolean ones. So we may guess that by using a classical boolean CSP solver to solve the abstract version, the abstraction method would result even more convenient.

Our results do not say anything about the convenience of our methodology on other classes of soft constraints. We plan to study the applicability of the abstraction methodology to solve also other classes of soft CSPs. It would be interesting also to study the interaction between the described solving methodology based on abstraction and the notion of global constraints.

Acknowledgements

This work was partially supported by the ASI project ARISCOM and by the Italian MIUR project COVER.

References

1. S. Bistarelli, P. Codognet, Y. Georget, and F. Rossi. Abstracting soft constraints. In K. Apt, E. Monfroy, T. Kakas, and F. Rossi, editors, *Proc. 1999 ERCIM/Compulog Net workshop on Constraints*, Springer LNAI 1865, 2000.
2. S. Bistarelli, P. Codognet, and F. Rossi. An Abstraction Framework for Soft Constraints, and Its Relationship with Constraint Propagation. In B. Y. Chouery and T. Walsh, Eds., *Proc. SARA 2000 (Symposium on Abstraction, Reformulation, and Approximation)*, Springer LNAI 1864, 2000.
3. S. Bistarelli, P. Codognet, and F. Rossi. Abstracting Soft Constraints: Framework, Properties, Examples. *Artificial Intelligence journal*, vol. 139, 2002.
4. Stefano Bistarelli, Thomas Fruhwirth, Michal Marte and Francesca Rossi. Soft Constraint Propagation and Solving in Constraint Handling Rules. In *Proc. ACM Symposium on Applied Computing, 2002*.
5. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.
7. D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*, pages 1131–1136. IEEE, 1993.
8. T. Frühwirth, Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming (P. J. Stuckey and K. Marriot, Eds.), *Journal of Logic Programming*, Vol 37(1-3):95-138 Oct-Dec 1998.
9. Yan Georget and Philippe Codognet. Compiling Semiring-based Constraints with clp(FD,S). In *Proc. 4th International Conference on Principles and Practice of Constraint Programming (CP98)*, pages 205–219, Springer-Verlag, LNCS 1520, 1998.
10. Ryszard Kowalczyk and Van Anh Bui. JFSolver: A Tool for Solving Fuzzy Constraint Satisfaction. In Hong Yan, Editor, *Proc. FUZZ-IEEE 2001 (10th IEEE International Conference on Fuzzy Systems)*, Melbourne, Australia, IEEE Press 2001.
11. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence Journal*, 8(1):99–118, 1977.
12. I. Pilan, F. Rossi. Abstracting soft constraints: some experimental results. *Proc. ERCIM/Colognet workshop on CLP and constraint solving*, Budapest, June 2003.
13. Zs. Ruttkay. Fuzzy constraint satisfaction. In *Proc. 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.
14. T. Schiex. Possibilistic constraint satisfaction problems, or "how to handle soft constraints?". In *Proc. 8th Conf. of Uncertainty in AI*, pages 269–275, 1992.

Tradeoff Generation Using Soft Constraints

Stefano Bistarelli[1,2], Jerome Kelleher[3], and Barry O'Sullivan[3]

[1] Istituto di Informatica e Telematica, CNR, Pisa, Italy
Stefano.Bistarelli@iit.cnr.it
[2] Dipartimento di Scienze
Universitá degli Studi "G. D'annunzio" di Chieti-Pescara, Italy
bista@sci.unich.it
[3] Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{j.kelleher,b.osullivan}@4c.ucc.ie

Abstract. Tradeoffs have been proposed in the literature as an approach to resolving over-constrainedness in interactive constraint-based tools, such as product configurators. It has been reported how tradeoffs can be modeled as additional constraints. This paper presents a formal framework for tradeoff generation based on the semiring approach to soft constraints. In particular, user preferences and tradeoffs are, respectively, represented as soft constraints and as an entailment operator. The entailment operator is used to interactively generate new constraints representing tradeoffs. The framework we present is well-motivated by real-world approaches that exploit tradeoff generation in online buying and configuration processes.

1 Introduction

A typical interactive configuration session is one where a human user articulates preferences for product features to a configurator which ensures consistency between the constraints of the problem and the user's desires. During such a session a point may be reached where all of the user's desires cannot be met. At this point the user could consider "tradeoffs" between his preferences. For example, in configuring a car, the user may find that it is impossible to have one "*with an engine size more that 2 litres and having a minimum fuel consumption of 15 km per litre*", but could accept a tradeoff: "*I will reduce my engine size requirement to 1.6 litres if I can have a minimum fuel consumption of 20 km per litre.*" Ideally, we would like the configurator to suggest appropriate tradeoffs to the user. There are a number of web-sites that attempt to help users to make tradeoffs between conflicting preferences and desires given the space of possible products that are available. The most well-known of these is the very successful *Active Buyer's Guide* web-site[1], which takes a similar two-stage approach to the one proposed here. Initially, user preferences are acquired with tradeoffs being proposed/elicited where necessary. Finally, users make their choices for the features of the desired product.

[1] See http://www.activebuyersguide.com

In this paper we extend and formalize previous work on tradeoffs. In [13] tradeoffs are crisp binary constraints that are interactively generated to substitute strict unary crisp constraints representing user desires. In this paper we increase the utility of tradeoff generation since the amount of information gathered from the user is increased by using soft constraints to represent preferences. Representing the preferences of the user in a formal way was not addressed in the earlier work in this area. Furthermore, the extended framework presented here is general enough to deal with any arity of preference constraints, not only unary ones as reported in [13].

The task of tradeoff generation is also formalized in this paper. The possible tradeoffs are given as the result of an "entailment" function. A filter function is used to select one of the possible entailed tradeoff constraints. The final generalization in our framework is that tradeoff constraints are not necessarily limited to being binary. This provides us with a richer model of tradeoff. For example, we may wish to add a non-binary tradeoff constraint in certain situations, such as when we are prepared to have a pair of constraints, c_x and c_y, replaced by a ternary constraint, $c'_{(x,y,z)}$. The additional constraining influence on z could be regarded as an *imposed* constraint.

To handle both of these extensions we use the semiring-based framework [4,5,7] that has been shown to be capable of representing both crisp and soft constraints in a uniform way.

Thus, the primary contribution of this paper is a formal and general theoretical framework for tradeoff generation for interactive constraint processing that uses soft constraints to represent user preferences and an entailment operator to generate tradeoffs.

The remainder of the paper, which extends [8], is organized as follows. Section 2 presents the necessary background on the semiring-based approach to handling soft constraints and on the tradeoff generation schema. Section 3 presents our general framework for tradeoff generation. An example is presented in Section 4. In Section 5 an evaluation of the framework is presented with a discussion of how this can be deployed in a real world context. Some concluding remarks are made in Section 6.

2 Background: Tradeoffs and Soft Constraints

Product configuration is becoming a well studied design activity which is often modeled and solved as a constraint satisfaction problem [1,12,14,19]. In this paper we present a formal framework for tradeoff generation in interactive constraint processing. In the existing literature on this topic, a tradeoff is a binary constraint which substitutes a pair of unary preference constraints; the tradeoff constraint representing a satisfactory compromise for the user [13]. For example, consider a set, $U = \{c_1, \ldots, c_k\}$, of user-specified unary preference constraints, and a set P of constraints defining the underlying problem, such that $U \cup P$ is inconsistent. A tradeoff constraint, T_{ij}, is a binary constraint involving a pair of variables, v_i and v_j, on which the user has specified a pair of unary preference constraints, $c_i \in U$ and $c_j \in U$, such that $U \cup T_{ij} - \{c_i, c_j\}$ is consistent and the user's preference on one variable has been strengthened and relaxed on the other. Therefore, currently, a tradeoff is a binary constraint which replaces a pair of unary preference constraints defined over the same pair of variables.

In this paper we regard each constraint in U as a *soft constraint* whose preference levels can be combined according to the specific notion of combination for the problem. Soft constraints associate a qualitative or quantitative value either to the entire constraint or to each assignment of its variables. Such values are interpreted as a level of preference, importance or cost. The levels are usually ordered, reflecting the fact that some levels (constraints) are *better* than others. When using soft constraints it is necessary to specify, via suitable combination operators, how the level of preference of a global solution is obtained from the preferences in the constraints.

Several formalizations of the concept of *soft constraints* are currently available. In the following, we refer to the formalization based on c-semirings [4,5,7], which can be shown to generalize and express both crisp and soft constraints [3].

A semiring-based constraint assigns to each instantiation of its variables an associated value from a partially ordered set. When dealing with crisp constraints, the values are the booleans *true* and *false* representing the admissible and non-admissible values; when dealing with soft constraints the values are interpreted as preferences.

The framework must also handle the combination of constraints. To do this one must take into account such additional values, and thus the formalism must provide suitable operations for combination ($\times$) and comparison ($+$) of tuples of values and constraints. This is why this formalization is based on the concept of c-semiring.

More precisely, they are based on a semiring structure $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and a set of variables V with domain D. In particular the semiring operation $\times$ is used to combine constraints together, and the $+$ operator for projection.

Technically, a *constraint* is a function which, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring that is $\mathcal{C} : \eta \to A$ is the set of all possible constraints that can be built starting from S, D and V (values in A are interpreted as level of preference or importance or cost).

Note that in this *functional* formulation, each constraint is a function (as defined in [7]) and not a pair (as defined in [5]). Such a function involves all the variables in V, but it depends on the assignment of only a finite subset of them. We call this subset the *support* of the constraint.

Consider a constraint $c \in \mathcal{C}$. We define his support as $supp(c) = \{v \in V \mid \exists \eta, d_1, d_2 . c\eta[v := d_1] \neq c\eta[v := d_2]\}$, where

$$\eta[v := d]v' = \begin{cases} d & \text{if } v = v', \\ \eta v' & \text{otherwise.} \end{cases}$$

Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the association $v := d_1$ (that is the operator $[\ ]$ has precedence over application).

When using soft constraints it is necessary to specify, via suitable combination operators, how the level of preference of a global solution is obtained from the preferences in the constraints. The combined weight of a set of constraints is computed using the operator $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ defined as $(c_1 \otimes c_2)\eta = c_1\eta \times_S c_2\eta$. Moreover, given a constraint $c \in \mathcal{C}$ and a variable $v \in V$, the *projection* of c over $V - \{v\}$, written $c \Downarrow_{(V-\{v\})}$ is the constraint c' s.t. $c'\eta = \sum_{d \in D} c\eta[v := d]$.

Informally, projecting means eliminating some variables from the support. This is done by associating to each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables. In short, combination is performed via the multiplicative operation of the semiring, and projection via the additive one.

The *solution* of a SCSP $P = \langle C, con \rangle$ is the constraint $Sol(P) = (\bigotimes C) \Downarrow_{con}$. That is, we combine all constraints, and then project over the variables in *con*. In this way we get the constraint with support (not greater than) *con* which is "induced" by the entire SCSP. Note that when all the variables are of interest we do not need to perform any projection.

Solutions are constraints in themselves and can be ordered by extending the $\leq_S$ order. We say that a constraint c_1 is at least as constraining as constraint c_2 if $c_1 \sqsubseteq c_2$, where for any assignment η of variables then $c_1 \sqsubseteq c_2 \equiv c_1\eta \leq_S c_2\eta$. Notice that using the functional formulation [7] we can also compare constraints that have different support (in [4,5] only constraints with the same support could be compared).

Sometimes it may be useful to find only a semiring value representing the least upper bound among the values yielded by the solutions. This is called the *best level of consistency* of an SCSP P and it is defined by $blevel(P) = Sol(P) \Downarrow_{\emptyset}$.

3 Tradeoff as an Entailment Operator

In this section we define a general notion of tradeoff using the semiring-based framework presented above. We use hard constraints to represent the strict and unmodifiable conditions of the problem (denoted P). We use soft constraints to represent, modifiable, user preferences (denoted U).

While, in general, we may have some constraints in P that are soft, we would regard this softness as a cost that would not be handled in the same way as user-specified preference constraints. In our framework we model softness in the user's desires, U, as preferences. The user makes statements like *"I prefer to have a petrol engine over a diesel one"* in a quantifiable manner by associating semiring values with each option. However, any softness in the physical constraints in the problem, P, represent the costs, or penalties, associated with relaxing them. These costs can be thought of as problem statements such as *"a diesel engine is not normally available for the small chassis, but for an additional cost, we can make the option available"*. Note that these types of softness are semantically different and we would treat them as such. For the remainder of the paper we will simply regard each problem constraint in P as a hard (crisp) constraint.

We model tradeoffs as a special *entailment operator* [20]. As shown in [7], an entailment operator for soft constraints, given a set of constraints C, generates constraints c' s.t. $\bigotimes C \sqsubseteq c'$ (written as $C \vdash c'$).

Tradeoffs specialize entailments in two respects:

– firstly, constraints c' generated by the tradeoff operator are *substituted* for the preference constraints, C, while entailed constraints are usually *added* to the problem.
– secondly, when we add tradeoffs, we do not necessarily obtain a globally better solution, but we may obtain a *Pareto incomparable* one. Specifically, while low preference constraints, $C \in U$, are substituted by c', thus increasing the overall level of preference, c' usually also lowers the preference of some other constraints $\bar{C} \in U$.

So, for instance, if $U = \{c_1, \ldots, c_n\}$, a tradeoff for constraints $C = \{c_1, c_2\}$ could be a constraint c' s.t. $C \vdash c'$ and $supp(c') \supseteq supp(c_1) \cup supp(c_2)$ (usually we have $supp(c') \supset supp(c_1) \cup supp(c_2)$). Formally, we can define the notion of potential tradeoffs as follows:

Definition 1 (Potential Tradeoffs). *Given a configuration problem $\{P \cup U\}$ and a subset of user preference constraints $C \subseteq U$. We say that c' is a* Potential Tradeoff *for C ($c' \in Trades_{\langle P,U \rangle}(C)$) if*

– *$supp(\bigotimes C) \subseteq supp(c')$; let's call $\bar{C} \subseteq U$ the greatest set of preference constraints s.t. $supp(\bigotimes\{C, \bar{C}\}) = supp(c')$;*
– *$C \vdash c'$ (that is $C \sqsubseteq c'$);*

The meaning of this definition is that a potential tradeoff will increase the level of some user preference constraints (those in $C = \{c_1, c_2\}$), and possibly lower some other ones whose support is in $\bar{C}$.

Notice that after the application of the tradeoff operator, we never obtain a best level of consistency worse than before.

Theorem 1. *Given a configuration problem $\{P \cup U\}$ and a subset of user preference constraints $C \subseteq U$. If c' is a* Potential Tradeoff *for C ($c' \in Trades_{\langle P,U \rangle}(C)$) Then, $blevel(P \cup U - C \cup \{c'\}) \not< blevel(P \cup U)$.*

3.1 Computing Tradeoffs

The way the preference constraints $C \subseteq U$ are selected, and the way a specific tradeoff c' is filtered from all the potential ones, and thus, $\bar{C}$ computed, is one of the most important issues when dealing with configuration problems. The potential tradeoffs can be restricted in various ways, which may be problem or context-specific. For example, we may wish to select a tradeoff in a way which could be regarded as "user-friendly". In the context of an interactive constraint-based configuration tool we may need to ensure that the user "trusts" the configurator, which may imply that previously accepted tradeoffs are not revisited. Some possible approaches to selecting preferences and filtering tradeoff constraints are presented in Section 3.2.

Notice that the filtered tradeoff could depend on the presence of a particular (partial) assignment, η, of the variables of the problem whose association has to be maintained[2].

Usually the configuration process first requests preference constraints from the user, and only if a solution better than a threshold α cannot be found with the acquired constraints, then a tradeoff is computed. To perform this check we always compute $blevel(P \cup U) = Sol(P \cup U) \Downarrow_{\emptyset}$ and compare this value with the semiring level α.

Therefore, tradeoff generation can be characterized in this context as a function:

$$Trades^{\alpha}_{\langle P,U,\eta,!\rangle} : \mathcal{C} \to \mathcal{C}$$

where:

- P is the set of *Problem constraints*;
- U is the set of *User-specified preference constraints*;
- ! is a filter (cut function) that first selects a set of preference constraints, $C \subseteq U$, and then selects one from among the potential tradeoffs giving c' (see Section 3.2).
- η is a (partial) assignment of the variables whose association has to be maintained;
- α represents the minimum best level of consistency we wish to achieve in the SCSP. The level α can be seen as the minimum level of global preference satisfaction we want to achieve;

Definition 2 (Tradeoffs). *Given a configuration problem $\{P \cup U\}$, and a subset of the user's preference constraints, $C \subseteq U$. We say that c' is a* Tradeoff *for C using the threshold α, the filter ! and the partial assignment η ($c' \in Trades^{\alpha}_{\langle P,U,\eta,!\rangle}(C)$), if the following are satisfied:*

- *c' is a* Potential Tradeoff, *that is:*
  - *$supp(\bigotimes C) \subseteq supp(c')$;*
  - *$C \vdash c'$;*
- *$blevel(\{P \cup U\}) < \alpha$ (i.e. the problem is no longer α-consistent);*
- *$blevel(\{P \cup \{U - C\} \cup c'\}) \not< \alpha$ (starting from a solution with an insufficient level of consistency, we want an assignment that gives a solution with a best level of consistency not worse than α).*
- *! is used as a filter (see Section 3.2).*

[2] In this paper η is fixed and never changes, so in all the definitions it could be omitted. Nevertheless it is important to include it in the notion of preference/tradeoff because in the next step of a configuration problem it will play an important role. After giving the user the opportunity to specify constraint preferences, he will be asked to make some more precise choices. In that phase of the configuration process, η becomes a partial assignment. We plan to address this second phase of the configuration problem as part of our research agenda in this area.

3.2 Heuristics for Selecting Preference and Tradeoff Constraints

To completely define a tradeoff function we need to specify the selection and filtering heuristic, !, we will use to:

1. select the user preference constraints to eliminate, C – we will denote this selector $!^{out}$, and
2. select a tradeoff, c', from the set of potential tradeoffs computed by the entailment operator – we will denote this filter $!^{in}$; notice that this also implies the selection of the preference constraints $\bar{C}$ whose level of preference could be reduced.

Before presenting some specific examples of the selection heuristic, $!^{out}$, and the filtering heuristic, $!^{in}$, recall that the trigger for generating tradeoff constraints is the detection that there does not exist a solution which is α-consistent, i.e. that $\mathit{blevel}(\{P \cup U\}) < \alpha$. In a configuration context α can represent a threshold in utility, cost or some other application-specific metric for measuring the quality or suitability of the outcome.

Below, we give here some possible instantiations of $!^{out}$ and $!^{in}$. Firstly we consider heuristics for implementing the $!^{out}$ selector. Of course, in addition to the heuristics we present below, a variety of others are possible which can be made application- or user-specific.

Random Selection: Random selection is always a possibility. In this case, the set C containing the preference constraint(s) to remove is randomly selected from amongst those in $!^{out}(U) = \{c_j \in U : P \cup U - \{c_j\}$ is α-consistent$\}$; This heuristic is an obvious naive approach.

Strictly related to P: The preference constraint we want to modify is strictly connected to the problem definition. In this case, the set C containing the preference constraint(s) to remove is selected from amongst those in $!^{out}(U) = \{c_j \in U : P \cup U - \{c_j\}$ is α-consistent $\wedge \exists c_i \in P : \mathit{supp}(c_i) \cap \mathit{supp}(c_j) \neq \emptyset\}$;

Has not appeared in a tradeoff: The preference constraint we want to modify has not already been affected by a tradeoff. In this case, the set C containing the preference constraint(s) to remove is selected from amongst those in $!^{out}(U) = \{c_j \in U : P \cup U - \{c_j\}$ is α-consistent $\wedge \nexists$ a tradeoff $t \in U : \mathit{supp}(t) \supseteq \mathit{supp}(c_j)\}$; this heuristic can be regarded as "user-friendly" since we do not ask the user to consider tradeoffs on variables which have already been involved in a tradeoff.

The tradeoff constraint, c', that we will choose to proposed to the user, by filtering from the set of potential tradeoffs using $!^{in}$, has the properties that it will:

1. reflect a relaxation of the user's preference for constraint(s) C (selected by using $!^{out}$), and
2. a strengthening of the user's preference for constraint(s) $\bar{C}$.

Proposition 1. *Given a configuration problem $\{P \cup U\}$, a subset of user preference constraints $C \subseteq U$ and a tradeoff c' for C; let also $\bar{C} \subseteq U$ the greatest set s.t. $supp(\bigotimes\{C, \bar{C}\}) = supp(c')$; Then,*

- $(P \cup \{U - C\} \cup c') \Downarrow_{supp(C)} \sqsupseteq (P \cup U) \Downarrow_{supp(C)}$;
- $(P \cup \{U - C\} \cup c') \Downarrow_{supp(\bar{C})} \sqsubseteq (P \cup U) \Downarrow_{supp(\bar{C})}$.

We now give some examples of $!^{in}$ filters that select one tradeoff from among all possible potential tradeoffs. Some possible approaches we could adopt, based on [13] are:

Maximum viability *c' is maximal w.r.t. P, U and η (that is for all $c'' \in Trades^{\alpha}_{\langle P,U,\eta,!\rangle}$ we have $\bigotimes\{P \cup \{U - C\} \cup c''\}\eta \not\geq \bigotimes\{P \cup \{U - C\} \cup c'\}\eta$;*

Minimum viability *c' is minimal w.r.t. P, U and η (that is there not exists $c'' \in Trades^{\alpha}_{\langle P,U,\eta,!\rangle}$ s.t. $\bigotimes\{P \cup \{U - C\} \cup c''\}\eta \leq \bigotimes\{P \cup \{U - C\} \cup c'\}\eta$.*

Notice that the first approach will be less tasking on the configurator since it always selects the less restrictive tradeoff, i.e. we will have $c' \Downarrow_{supp(C)} = \mathbf{1}$; the tradeoff will give to all domain values of the variables in C the best preference. However, in this way the preferences made by the user on this assignment are lost. Essentially, using the first approach we maximize the possible solutions that the user will be able to choose from, but we cannot guarantee that the user's original set of preferences will be satisfied to the maximal extent.

On the other hand, the second approach will try to always stay as close as possible to the user's preferences, i.e. we will have $c' \Downarrow_{supp(C)} \sqsupset C$. The tradeoff will increase the preference on C just sufficiently to reach the required level of consistency α. Therefore, such a minor increment could result in a significant number of tradeoff interactions during the configuration process. In fact, the constraint c' inserted by the configurator could be too strict to be α-consistent when the user will insert new preference constraints in future interactions. However, in this case we run the risk of being misled by the user, in the sense that we may have to generate many tradeoffs in order to recover from a set of strong and over-constraining preferences.

It is worth pointing out at this point that a good user-interface could assist in the detection of preference constraints to strengthen and which to relax. An appropriate user-interface could also take care of preference elicitation. For example, we could assume that importance relationships between variables is reflected by the order in which user-choices are made. This is also an issue we are investigating as part of our research agenda in this area.

Furthermore, the approach we are advocating in this work can be seen as an automated version of that used on the very successful *Active Buyer's Guide* web-site.

4 An Example

The example configuration problem that will be studied here is based on the soft n-queens problem [6]. However, we will assume, for space reasons, that all of the problem constraints in P are hard in the traditional sense. Furthermore, for this example we have chosen the fuzzy semiring, $S = \langle [0,1], max, min, 0, 1 \rangle$, so constraints in U have preference between 0 and 1 and they are combined using min.

The n-queens problem exhibits many of the features one finds in a real-world configuration problem. Specifically, consider the formulation of the problem that we use here, where each column is represented as a variable and the row position of a queen in a particular column is denoted by the value assigned to the appropriate variable. Between the variables are constraints encoding the requirement that no pair of queens attack each other. In terms of a configuration problem, the variables represent the features over which the user/customer make choices. The domain of each variable represents the possible choices that the user can make for a particular feature. The constraints between the variables represent the compatibilities between combinations of instantiations of the features of the product.

The user attempts to solve this configuration problem by interactively specifying some preference constraints to a constraint-based configurator. During the interactive session with the configurator, the user may specify a preference constraint which causes the problem to become "over-constrained", identified by the problem becoming less than α-consistent (for some fixed α). At this point our configurator attempts to recommend tradeoff constraints to the user which he/she can accept before continuing.

Thus, the interactive solving process, based on [13], can be summarized as follows:

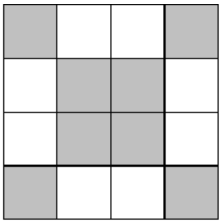
```
Repeat until preferences are specified for all variables in the problem:
  – Repeat until over-constrained – $blevel(P \cup U) < \alpha$:
      • the user specifies a preference constraint, $c \in U$;
  – Repeat until user is satisfied:
      • the system proposes a tradeoff
```

For the purposes of this example let's assume we wish to solve the 4-Queens problem with a crisp set of problem constraints, P, i.e. the configuration problem constraints are hard. This means that the only preference/costs we have to consider are given by the user (inside the set U).

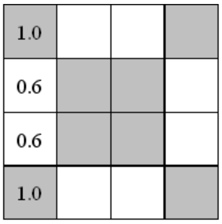
In the following example we will consider only unary preference constraints representing the columns, $\{c_1, c_2, c_3, c_4\}$ of the chess-board. The user proposes unary preference constraints on each of the columns, sequentially from the first to the fourth, representing wishes for the placement of queens.

Furthermore, we also assume that only binary tradeoff constraints c' will be generated. Finally, lets assume that the user wishes to achieve a minimum level of 0.5-consistency, i.e. $\alpha = 0.5$.

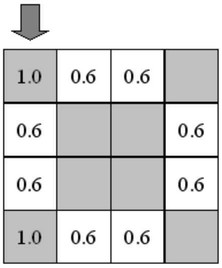
Problem constraints in *P*: Figure 1(a) the *no-attack* crisp constraints in$P$ are represented. The grey squares represent impossible configurations (i.e. with preference level 0); the white squares represent consistent positions for the queens.



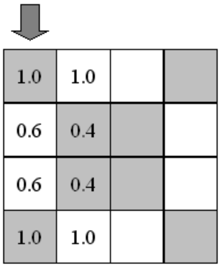
(a) Chess-board with crisp xconstraints.



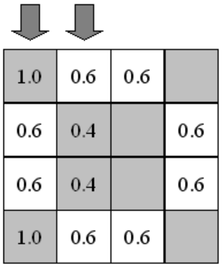
(b) Preferences for the 1st column.



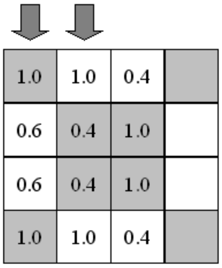
(c) Evaluation of the 1st preference.



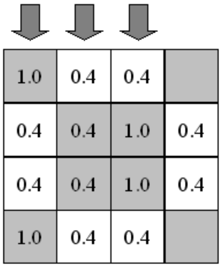
(d) Added preferences for the 2nd column.



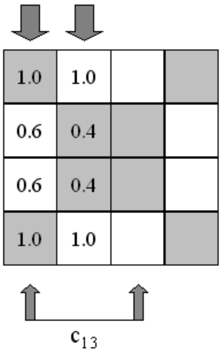
(e) Evaluation of the preference for the 1st and 2nd columns.



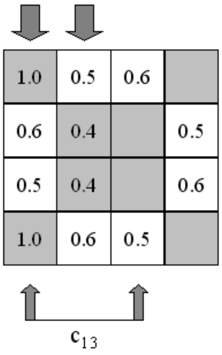
(f) Added preferences for the 3rd column.



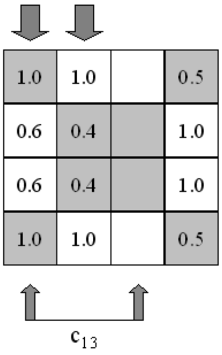
(g) Evaluation of the preferences for 1st,2nd and 3rd columns.



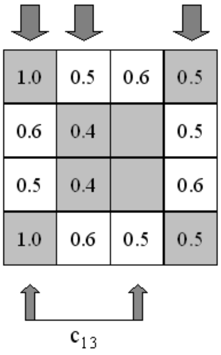
(h) Added a tradeoff between preferences on the 1st and 3rd column.



(i) Evaluation of the preferences for 1st and 2nd columns and of the tradeoff.



(j) Added preferences for the 4th column.



(k) Evaluation after the preferences over the 4th column.

Fig. 1. An example interaction with all hard constraints in P.

User Decision #1: The user states a unary preference constraint on the values for column 1: $c_1(1) = 1.0, c_1(2) = 0.6, c_1(3) = 0.6, c_1(4) = 1.0$ (see Figure 1(b)). The constraint network representing this problem is still α-consistent, because it is possible to achieve a solution having a semiring value of 0.6 (Figure 1(c)) by putting the queen in row 2 or 3, so the user is free to continue articulating preference constraints. Notice that queens cannot be positioned in row 1 or row 4 because the crisp constraints in P do not admit any solution with queens in these positions.

User Decision #2: The user states a unary preference constraint on the values for column 2: $c_2(1) = 1.0, c_2(2) = 0.4, c_2(3) = 0.4, c_2(4) = 1.0$ (Figure 1(d)). The constraint network representing this problem is still α-consistent: setting $c_1 = 2$ and $c_2 = 4$ could yield a solution with preference 0.6 (Figure 1(e)). In fact, possible solutions are obtained by putting the queen in row 1 or 4 [3] Therefore, the user is free to continue articulating preferences.

User Decision #3: The user states a unary preference constraint on the values for column 3: $c_3(1) = 0.4, c_3(2) = 1.0, c_3(3) = 1.0, c_3(4) = 0.4$ (Figure 1(f)). The constraint network representing this problem is no longer α-consistent: the best solution has a semiring value of 0.4 (for example setting $c_1 = 2$, $c_2 = 4$ and $c_3 = 1$) (Figure 1(g)). Therefore, the user must consider tradeoffs.

Tradeoff #1: The configurator needs to select a preference constraint to remove and a tradeoff to add. In this case all the heuristics presented in Section 3.2 for $!^{out}$ will select the preference constraint on column 3. In fact $C = \{c_3\}$ is the only set of preference constraint s.t. $P \cup U - C$ is 0.5-consistent.

Furthermore, lets assume that the tradeoff constraint selected using $!^{in}$ involves column 1 and 3. Let's suppose our heuristic select: $c_{13} = \{(1,1) = 0.6, (2,1) = 0.6, (3,1) = 0.6, (4,1) = 0.6, (1,2) = 1.0, (2,2) = 1.0, (3,2) = 1.0, (4,2) = 1.0, (1,3) = 1.0, (2,3) = 1.0, (3,3) = 1.0, (4,3) = 1.0, (1,4) = 0.5, (2,4) = 0.5, (3,4) = 0.5, (4,4) = 0.5\}$ (Figure 1(h)).

Assuming the user accepts this tradeoff constraint, the network is once again α-consistent (Figure 1(i)). The user could set $c_1 = 2$, $c_2 = 4$ and $c_3 = 1$ for a semi-ring value of 0.5. Note that by adding the tradeoff we also remove the unary preference constraint on column 3 and relax the preferences on column 1.

User Decision #4: The user states a unary preference constraint on the values for column 4: $c_4(1) = 0.5, c_4(2) = 1.0, c_4(3) = 1.0, c_4(4) = 0.5$ (Figure 1(j)). The constraint network representing this problem is still α-consistent (Figure 1(k)). The user could set $c_1 = 2$, $c_2 = 4$, $c_3 = 1$ and $c_4 = 3$ for a semi-ring value of 0.6. Preferences have now been defined for each column.

[3] In this example all of the constraints in P are crisp, so the possible positions of the queens are strictly imposed. Instead, if P contained soft constraints, more possibilities (with different costs) would be possible.

5 Evaluation and Discussion

To inform intuition on our approach to tradeoff generation based on the framework presented in this paper, we implemented a concretisation which satisfies the requirements of the framework in a minimal way. Specifically, we generated random entailed constraints c' such that $\forall\eta.c'\eta \geq \alpha$. In this way we guarantee that the problem will always be alpha-consistent after we perform the tradeoff operation. Obviously this approach is not intended to reflect any "real-world" tradeoffs, however, it does provide us with some basic insights as to the number of tradeoffs will be required with different levels of α in a real-world instantiation of this framework.

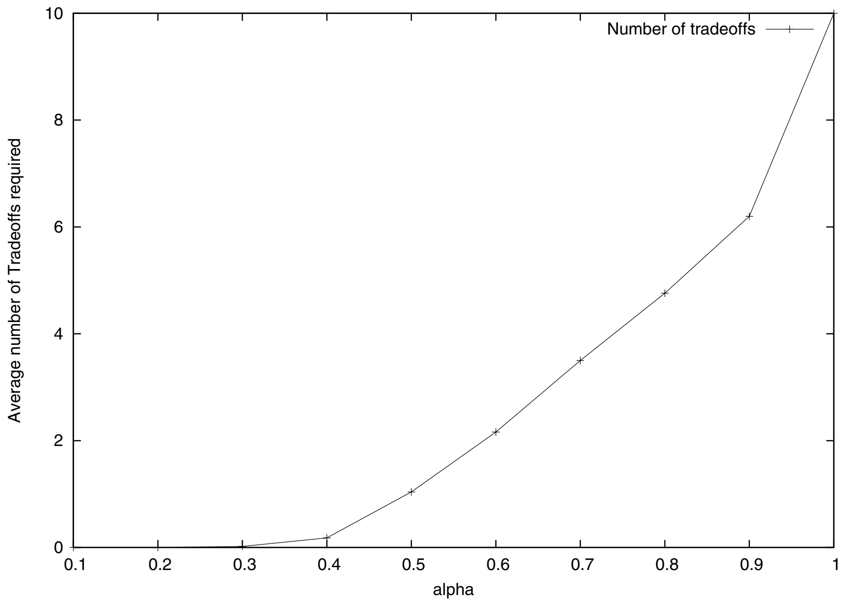


Fig. 2. Average number of tradeoffs required over a fifty user-interaction simulations at various levels of α.

In Figure 2 we show results obtained by averaging the number of tradeoffs required over fifty user-interaction simulations. In this experiment, we use the 10-queens problem. The constraints that define this problem represent the crisp set of constraints P. We randomly generate unary fuzzy constraint functions to represent each of the user preference constraints in U. To simulate a general user-interaction we take a sequential approach to eliciting user preferences and determine if the user has stated a preference which over-constrains the overall problem (i.e. $P \cup U$). If the user has over-constrained, identified by the problem becoming α-inconsistent, then it is necessary to generate a tradeoff constraint.

Thus, in this experiment we sequentially generate 10 unary preference functions and evaluate $blevel(P \cup U)$ after each constraint is added. If this evaluates to greater than α, we continue on to generate the next preference function, if not we generate a tradeoff to make the problem α-consistent again.

If we consider Figure 2 we see that at low levels of α we do not need to generate many tradeoffs, but at higher levels of α, as the user gets more demanding, we find that the number of tradeoffs required to find a solution increase dramatically. This is to be expected since at lower levels of α the likelihood of over-constraining the problem are very low, while as α increases we are more likely to over-constrain the problem after each user preference constraint is added to the model.

An interesting question here is how many α-consistent solutions of the problem do we find as α changes. Interestingly, we observed that at very low and very high levels of α we "lost" very few α-consistent solutions. By the term "lost" solution, we mean how many of the solutions to the problem are no longer α-consistent, at the end of the simulated interaction, due to tradeoffs. Specifically, we observed that as α increased, the number of solutions we "lost" gradually increased from 0 to some maximum, less than the number of solutions to the problem, and after some level of α began to decrease again. From Figure 2 we can see that low (high) α implies fewer (more) tradeoffs. This suggests that having to rely either lightly or heavily on tradeoffs is likely to be a successful strategy to find satisfactory solutions to a problem. Of course the extent to which this is possible is determined by the level of α sought by the user. Therefore, we are more likely to be able to satisfy the expectations of easy-to-please users, who require low α, or very selective users, who require high α. Note that this is a similar phenomenon to that observed by Freuder and O'Sullivan in the earlier work on tradeoff generation for crisp CSPs [13]. They noted how both greedy and easy-to-please users were easy to satisfy, but non-committal users were difficult.

While the instantiation of the tradeoff framework for soft constraints studied in this evaluation is based on very naive random methods to generating user preferences and tradeoff constraints, we can regard the results presented in Figure 2 as a worst-case scenario for this problem. While we always succeed in finding a solution, any form of intelligence that can be built into the selection heuristics, $!^{out}$ and $!^{in}$, will improve upon these results. This raises the interesting issue of how our work can be deployed in a real-world context. We address that issue in some detail below.

Many approaches to dealing with compromises in user preferences or constraints exist [2,9,10,16,17,18]. In our work we are concerned with attempting to automate tradeoff generation in order to take the onus from the user and place it with the system. The objective is that users will be assisted find satisfactory tradeoffs to their preference constraints in a search-style interaction while the system monitors consistency. Ideally, the tradeoff generation mechanism in such an interactive system will be capable of presenting tradeoffs which are likely to be accepted by the user, so that he or she can continue to search for his or her solution. This style of interaction is search focused, rather than the solution focused approaches of Recommender Systems or example critiquing systems [11].

In the framework we present here, the critical components are the filter used to select constraints to relax to regain consistency ($!^{out}$) and the filter used to decide what preferences to modify to form a tradeoff ($!^{in}$). One approach we could adopt is to attempt to learn appropriate filters based on past sessions with the user, or application-specific characteristics of the problem domain.

We also do not preclude user involvement in the tradeoff process. Indeed, one could imagine that the user participates in the selection of preferences to relax and strengthen. The framework we have presented still applies.

Finally, the role of user-interfaces for preference elicitation is an important one. For example, a user-interface can help translate user-choices into appropriate semiring values for each preference constraint. Also, a user-interface could provide us with a mechanism for reasoning about the relative importance of user preferences, upon which we could develop appropriate filters.

Therefore, we would argue that we have presented a generic framework for interactive search-based tradeoff systems. The framework is general purpose and can be instantiated in a number of ways. One direction for future work is the consideration of particular instantiations of the framework targeted at particular domains, or types of interaction.

6 Conclusions

Tradeoffs have been proposed in the literature as an approach to resolving over-constrainedness in interactive constraint-based tools, such as product configurators. It has already been reported in the literature how tradeoffs can be modeled as additional constraints. This paper presents a formal framework for tradeoff generation based on the semiring approach to handling soft constraints. In particular, we present a formal and general definition of tradeoff generation for interactive constraint processing. The framework we present is well-motivated by real-world approaches that exploit tradeoff generation in online buying and configuration processes.

Our research agenda in this area involves studying intelligent interfaces for reasoning about the relative importance of the user's preferences. For example, we could assume that importance relationships between variables are reflected by the order in which user-choices are made. We are also interested in a detailed empirical evaluation of a number of strategies for learning appropriate filter functions for more natural user-friendly interaction.

In summary, we have presented a formal framework for studying a very important aspect of interactive constraint processing, the ability to assist users achieve their desires to the maximal degree possible. This framework provides the basis for a research agenda in the area of interactive constraint satisfaction with practical applications in domains such as product configuration, e-commerce, interactive scheduling, negotiation and explanation. As future work we also plan to integrate the notion of tradeoff into the CHR framework [15].

Acknowledgment

This work has received support from Enterprise Ireland under their Basic Research Grant Scheme (Grant Number SC/02/289) and their International Collaboration Programme (Grant Number IC/2003/88).

References

1. J. Amilhastre, H. Fargier, and P. Marguis. Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artificial Intelligence*, 135:199–234, 2002.
2. D. Bahler, C. Dupont, and J. Bowen. Mixed quantitative/qualitative method for evaluating compromise solutions to conflict in collaborative design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 9:325–336, 1995.
3. S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and Valued CSPs: Frameworks, properties, and comparison. *CONSTRAINTS: An international journal. Kluwer*, 4(3), 1999.
4. S. Bistarelli, U. Montanari, and F. Rossi. Constraint Solving over Semirings. In *Proc. IJCAI95*, San Francisco, CA, USA, 1995. Morgan Kaufman.
5. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, Mar 1997.
6. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Logic Programming: Syntax and Semantics. *ACM Transactions on Programming Languages and System (TOPLAS)*, 23:1–29, 2001.
7. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. In *Proc. 11th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science (LNCS), pages 53–67. Springer, 2002.
8. S. Bistarelli and B. O'Sullivan. A theoretical framework for tradeoff generation using soft constraints. In *Proceedings of AI-2003*, pages 69–82. Springer, 2003.
9. Ronen I. Brafman and Carmel Domshlak. Tcp-nets for preference-based product configuration. In *Proceedings of the Forth Workshop on Configuration (in ECAI-02)*, pages 101–106, July 2002.
10. Yannick Descotte and Jean-Claude Latombe. Making compromises among antagonist constraints in a planner. *Artificial Intelligence*, 27:183–217, 1985.
11. B. Faltings, P. Pu, M. Torrens, and P. Viappiani. Designing example-critiquing interaction. In *International Conference on Intelligent User Interfaces*, 2004.
12. A. Felfernig, G. Friedrich, D. Jannach, and M. Stumpter. Consistency-based diagnosis of configuration knowledge-bases. In *Proceedings of the 14h European Conference on Artificial Intelligence (ECAI'2000)*, pages 146–150, 2000.
13. E. C. Freuder and B. O'Sullivan. Generating tradeoffs for interactive constraint-based configuration. In *Proceedings of CP-2001*, pages 590–594, Nov 2001.
14. E.C. Freuder, C. Likitvivatanavong, M. Moretti, F. Rossi, and R.J. Wallace. Computing explanations and implications in preference-based configurators. In Barry O'Sullivan, editor, *Recent Advances in Constraints*, volume 2627 of *LNAI*, pages 76–92, 2003.
15. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science (LNCS)*, pages 90–107. Springer, 1995.

16. R.L. Keeney and H. Raifa. *Decisions with Multiple Objectives: Preferences & Value Tradeoffs*. Cambridge University Press, 2002.
17. P. Pu, B. Faltings, and P. Kumar. User-involved tradeoff nalysis in configuration tasks. In *Proceedings of the Third CP Workshop on User-Interaction in Constraint Satisfaction*, pages 85–102, Septembet 2003.
18. S. Y. Reddy, K. W. Fertig, and D. E. Smith. Constraint management methodology for conceptual design tradeoff studies. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conference*, August 1996. Irvine, California.
19. D. Sabin and R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems and their applications*, 13(4):42–49, July–August 1998. Special Issue on Configuration.
20. V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

Experimental Evaluation of Interchangeability in Soft CSPs

Nicoleta Neagu[1], Stefano Bistarelli[2,3], and Boi Faltings[1]

[1] Artificial Intelligence Laboratory (LIA), Computer Science Department, EPFL
CH-1015, Ecublens, Switzerland
{boi.faltings,nicoleta.neagu}@epfl.ch

[2] Dipartimento di Scienze, Universitá "D'Annunzio"
Viale Pindaro 87, I-65127 Pescara, Italy
bista@sci.unich.it

[3] Istituto di Informatica Telematica (IIT), CNR,
Via G. Moruzzi 1, I-56124, Pisa, Italy
Stefano.Bistarelli@iit.cnr.it

Abstract. In [8], Freuder defined interchangeability for classical Constraint Satisfaction Problems (CSPs). Recently [2], we extended the definition of interchangeability to *Soft* CSPs and we introduced two notions of relaxation based on degradation δ and on threshold α ($^{\delta}$neighborhood interchangeability ($^{\delta}NI$)and $_{\alpha}$neighborhood interchangeability ($_{\alpha}NI$)). In this paper we extend the study introduced in [11] and we analyze the presence of the relaxed version of interchangeability in random soft CSPs. We give a short description of the implementation we used to compute interchangeabilities and to make the tests. The experiments show that there is high occurrence of $_{\alpha}NI$ and $^{\delta}NI$ interchangeability around optimal solution in fuzzy CSPs and weighted CSPs. Thus, these algorithms can be used successfully in solution update applications. Moreover, it is also showed that *NI* interchangeability can well approximate full interchangeability (*FI*).

Keywords: soft constraint satisfaction problems, substitutability, interchangeability.

1 Introduction

Interchangeability in constraint networks has been first proposed by Freuder [8] to capture equivalence among the values of a variable in a discrete constraint satisfaction problem. Value $v = a$ is *substitutable* for $v = b$ if for any solution where $v = a$, there is an identical solution except that $v = b$. Values $v = a$ and $v = b$ are *interchangeable* if they are substitutable both ways. *Full Interchangeability* considers all constraints in the problem and checks if values a and b for a certain variable v can be interchanged without affecting the global solution. The localized notion of *Neighbourhood Interchangeability* considers only the constraints involving a certain variable v. Interchangeability has since found other applications in abstraction frameworks [10, 16, 6] and solution adaptation [15].

One of the difficulties with interchangeability has been that it does not occur very frequently.

In many practical applications, constraints can be violated at a cost, and solving a CSP thus means finding a value assignment of minimum cost. Various frameworks for solving such soft constraints have been proposed in [9, 7, 13, 14, 3]. The soft constraints framework of c-semirings [3] has been shown to express most of the known variants through different instantiations of its operators, and this is the framework we are considering in this paper.

In [2] we extended the notion of interchangeability to Soft CSPs. The most straightforward generalization of interchangeability to Soft CSPs would require that exchanging one value for another does not change the quality of the solution at all. Nevertheless, this generalization is likely to suffer from the same weaknesses as interchangeability in hard CSPs, namely that it is very rare.

Fortunately, soft constraints also allow weaker forms of interchangeability where exchanging values may result in a degradation of solution quality by some measure δ. By allowing more degradation, it is possible to increase the amount of interchangeability in a problem to the desired level. The ${}^{\delta}$substitutability and ${}^{\delta}$interchangeability concepts ensure this quality. This is particularly useful when interchangeability is used for solution adaptation. Another use of interchangeability is to reduce search complexity by grouping together values that would never give a sufficiently good solution. In ${}_{\alpha}$substitutability/interchangeability, we consider values interchangeable if they give equal solution quality in all solutions better than α, but possibly different quality for solutions whose quality is $\leq \alpha$.

The behaviour of *NI* sets in the Soft CSP frameworks is still unexploited. For this motivation we study and evaluate here how *NI* behaves in soft CSPs frameworks (mainly fuzzy and weighted CSPs).

In the following we first remind some details about Interchangeability and soft Constraint Satisfaction Problems. We give also some details about the java implementation [11] we used to compute to ${}^{\delta}/_{\alpha}$substitutability/interchangeability. Central results of the paper are described in Section 4 where the results of some tests are described. Conclusions and possible future work are presented in the last section.

2 Soft CSPs

A soft constraint may be seen as a constraint where each instantiations of its variables has an associated value from a partially ordered set which can be interpreted as a set of preference values. Combining constraints will then have to take into account such additional values, and thus the formalism has also to provide suitable operations for combination ($\times$) and comparison ($+$) of tuples of values and constraints. This is why this formalization is based on the concept of c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, which is just a set A plus two operations[1].

[1] In [3] several properties of the structure are discussed. Let us just remind that it is possible to define a partial order $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$.

Constraint Problems. Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V over a finite domain D, a *constraint* is a function which, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring.

By using this notation we define $\mathcal{C} = \eta \to A$ as the set of all possible constraints that can be built starting from S, D and V. Consider a constraint $c \in \mathcal{C}$. We define his support as $supp(c) = \{v \in V \mid \exists \eta, d_1, d_2 . c\eta[v := d_1] \neq c\eta[v := d_2]\}$, where

$$\eta[v := d]v' = \begin{cases} d & \text{if } v = v', \\ \eta v' & \text{otherwise.} \end{cases}$$

Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the association $v := d_1$ (that is the operator [ ] has precedence over application).

Combining Soft Constraints. Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times_S c_2\eta$.

In words, combining two constraints means building a new constraint whose support involve all the variables of the original ones, and which associates to each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate subtuples.

Interchangeability. In soft CSPs, there are not any crisp notion of consistency. In fact, each tuple is a possible solution, but with different level of preference. Therefore, in this framework, the notion of interchangeability become finer: to say that values a and b are interchangeable we have also to consider the assigned semiring level.

More precisely, if a domain element a assigned to variable v can be substituted in each tuple solution with a domain element b without obtaining a worse semiring level we say that b is full substitutable for a (that is $b \in FS_v(a)$ if and only if $\bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b]$). When we restrict this notion only to the set of constraints C_v that involve variable v we obtain a local version of substitutability (that is $b \in NS_v(a)$ if and only if $\bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b]$).

When the relations hold in both directions, we have the notion of *Full and Neighbourhood Interchangeability* of b with a.

This means that when a and b are interchangeable for variable v they can be exchanged without affecting the level of any solution.

Extensivity ($NI_v(a/b) \implies FI_v(a/b)$) and transitivity ($b \in NS_v(a), a \in NS_v(c) \implies b \in NS_v(c)$) of interchangeability can be used to define an algorithm able to compute a subset of the interchangeabilities. When the times operator of the semiring is idempotent the algorithm, instead of considering the combination of all the constraint C_v involving a certain variable v, can check interchangeability on each constraint itself [2] giving raise to a low complexity bound.

Algorithm 1 shows the algorithm that can be used to find domain values that are Neighbourhood Interchangeable. It uses a data structure similar to the

```
1: Create the root of the discrimination tree for variable $v_i$
2: Let $C_{v_i} = \{c \in C \mid v_i \in supp(c)\}$
3: Let $D_{v_i} = \{$the set of domain values $d_{v_i}$ for variable $v_i\}$
4: for all $d_{v_i} \in D_{v_i}$ do
5:    for all $c \in C_v$ do
6:       execute Algorithm $NI$-Nodes$(c, v, d_{v_i})$ to build the nodes associated with $c$
7:    Go back to the root of the discrimination tree.
```

Algorithm 1: Algorithm to compute neighbourhood interchangeable sets for variable v_i.

discrimination trees, first introduced by Freuder in [8]. Algorithm 1 can compute different versions of neighbourhood interchangeability depending on the algorithm $NI - nodes$ used. Algorithm 2 shows the simplest version without threshold or degradation.

```
1: for all assignments $\eta_c$ to variables in $supp(c)$ do
2:    compute the semiring level $\beta = c\eta_c[v_i := d_{v_i}]$,
3:    if there exists a child node corresponding to $\langle c = \eta_c, \beta \rangle$ then
4:       move to it,
5:    else
6:       construct such a node and move to it.
7: Add $v_i, \{d_{v_i}\}$ to annotation of the last build node,
```

Algorithm 2: NI-Nodes(c, v, d_{v_i}) for Soft NI.

Degradations and Thresholds. In soft CSPs, any value assignment is a solution, but may have a very bad preference value. This allows broadening the original interchangeability concept to one that also allows degrading the solution quality when values are exchanged. We call this ${}^{\delta}$interchangeability, where δ is the *degradation* factor.

When searching for solutions to soft CSP, it is possible to gain efficiency by not distinguishing values that could in any case not be part of a solution of sufficient quality. In ${}_{\alpha}$interchangeability, two values are interchangeable if they do not affect the quality of any solution with quality better than α. We call α the *threshold* factor.

Both concepts can be combined, i.e. we can allow both degradation and limit search to solutions better than a certain threshold (${}^{\delta}_{\alpha}$interchangeability). By extending the previous definitions we obtain thresholds and degradation version of full/neighbourhood substitutability/interchangeability:

- we say that b is ${}^{\delta}$*Full Substitutable* for a on v ($b \in {}^{\delta}FS_v(a)$) if and only if for all assignments η, $\bigotimes C\eta[v := a] \times_S \delta \leq_S \bigotimes C\eta[v := b]$;
- we say that b is ${}_{\alpha}$*Full substitutable* for a on v ($b \in {}_{\alpha}FS_v(a)$) if and only if for all assignments η, $\bigotimes C\eta[v := a] \geq \alpha \implies \bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b]$.

Let consider a Fuzzy CSP example as in Figure 1. By applying the previous definitions we detect that for $\delta = 0.1$, values b and c for variable X_2 are $^{0.1}$*interchangeable.* When considering a lower degradation of $\delta = 0.4$, these values are not anymore $^{\delta}$*in t e rc han geable*

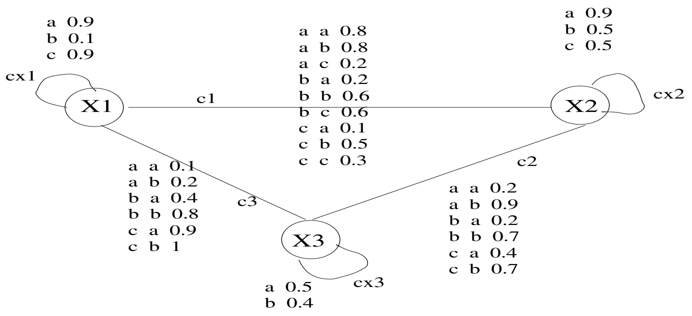


Fig. 1. Fuzzy CSP example.

By applying threshold definition for $\alpha = 0.4$ for the same values b and c of variable X_2, we can notice that these values are $_{0.4}$*interchangeable.*

3 The Java Implementation

We implemented the soft CSP module and the interchangeability module as an extension of the JCL, developed at the Artificial Intelligence Laboratory (EPFL) [4]. JCL is implemented in java which ensures portability on all the platforms. The library contains a CSP package describing constraint satisfaction problems and several solvers. In this section, we describe briefly the JCL and the CSP module.

The Java Constraint Library. The Java Constraint Library (JCL) is a library containing common constraint satisfaction techniques which provides services for creating and managing discrete CSPs and applying preprocessing and search algorithms.

We developed two new packages on top of JCL: the first one models and solve soft constraint satisfaction problems; the second one computes interchangeabilities for crisp and soft CSPs, see Figure 2.

The Soft_CSP Module. The Soft_CSP package extends the CSP class in order to support softness. We implemented the scheme from [3] where preferences levels are assigned both to variables values (implemented as soft unary constraints by the class `SoftUnaryConstraint`) and to tuples of values over the constraints. In particular, in the actual implementation we only consider binary constraints (by `SoftBinaryConstraint`), but the class can be easily extended in this direction.

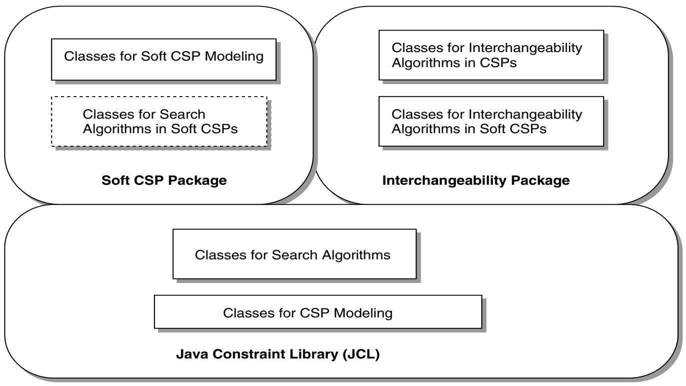

Classes for Soft CSP Modeling
Classes for Search Algorithms in Soft CSPs
Soft CSP Package
Classes for Interchangeability Algorithms in CSPs
Classes for Interchangeability Algorithms in Soft CSPs
Interchangeability Package
Classes for Search Algorithms
Classes for CSP Modeling
Java Constraint Library (JCL)

Fig. 2. Soft CSP and Interchangeability Modules on top of Java Constraint Language(JCL).

The Soft_CSP package supports classical, fuzzy, probabilistic and weighted CSPs by using appropriate semirings. The semiring class parameterizes the type of the CSP and the respective operations of constraints combinations (and projection).

The Interchangeability Module. This module implements all the algorithms for computing classical interchangeability for all the semiring types. It provides also the computational classes for degradation δ and/or threshold α interchangeability which finds ${}^{\delta}/_{\alpha}NI$ sets.

4 Test Evaluation

Occurrence of *NI* in classical CSP have been already studied to improve search [1], for resource allocation application [5] and for configuration problems [12]. One of the main result is that in problems of small density the number of *NI* sets increases with the domain size.

The behavior of *NI* sets in the Soft CSP frameworks is still unexploited. For this motivation we study and evaluate here how *NI* behaves in the Soft CSP framework.

We have done our experiments for fuzzy and weighted CSPs representing the important classes of Soft CSPs dealing with an idempotent and non-idempotent times operation respectively. The motivation for considering both classes comes from the fact that solving Soft CSP when the combination operation is not idempotent is extremely hard [3].

Usually the structure of a problem is characterised by four parameters:

- *Problem Size*: This is usually the number of its variables;
- *Domain Size*: The average of the cardinality of the domain of the variables;

– *Problem Density*: This value (measured on the interval [0,1]) is the ratio of the number of constraints relatively to the minimum and maximum number of allowed constraints in the given problem. Considering the constraint problem as a constraint graph $G = (V, E)$ where V represents the vertices (variables) (with $n := |V|$) and E edges (constraints) (with $e := |E|$); the density is computed as $denscsp = \frac{e - e\_min}{e\_max - e\_min}$, where $e\_min = n - 1$ and $e\_max = \frac{n(n-1)}{2}$;
– *Problem Tightness*: This measure is obtained as the average of tightness of all the constraints. For soft constraints we consider it as the ratio between the sum of the semiring values associated to all the tuples in all the constraints, and the value obtained by multiplying the **1** element of the semiring (that is the maximum) for the number of all possible tuple (that is the *constraint-number* $\times$ *domainsize*).

For both fuzzy and weighted CSPs we observed that the density and number of variables do not influence too much the occurrence of interchangeable values. There is instead a (weak) dependency from the domain size: *the number of interchangeabilities increases with the resources*. This result from the test is obvious when dealing with crisp CSPs, but for soft problems this could be not so obvious. We have instead found that the CSP tightness influence the occurrence of interchangeable values.

In the following we use the model of measuring *NI* sets developed in [5] with some adaptation needed in order to deal with softness. We report here the results for problem sizes n = 10, while varying the density $dens - csp \in \{0.1, 0.2, \ldots, 0.9\}$, the tightness $tightness - csp \in \{0.1, 0.2, \ldots, 0.9\}$ and the maximum domain size $dom - size = \{\frac{n}{10}, \frac{2n}{10}, \ldots, \frac{9n}{10}, n\}$. The semiring values are generated uniformly and accordingly to the CSP tightness. For each case, ten random problems were generated and then graphically represented by considering the measures $measure_\alpha NI$ and $measure^\delta NI$ described below.

In all the graphs we highlight where is the position of the optimal solution. In fact, when dealing with crisp CSP there is no notion of optimality, but for soft CSP each solution has an associated level of preference. It is important to study *NI* occurrence around optimal solutions because we are often interested to discard solutions of bad quality.

4.1 Fuzzy CSPs

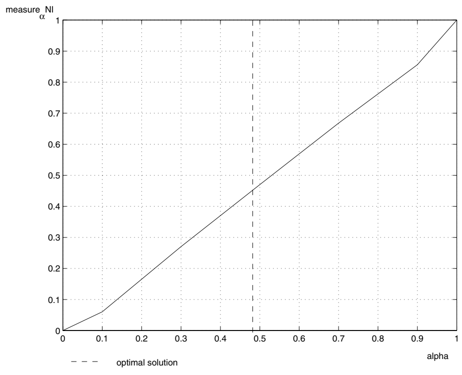
Fuzzy CSPs are a representative example of soft CSP based on an idempotent times semiring operator. Informally, for fuzzy CSPs the weights assigned to the tuples represents how much the tuple is satisfied, and the semiring operations are min for combination and max for projection.

$\delta/\alpha NI$ Occurrence in Fuzzy CSPs

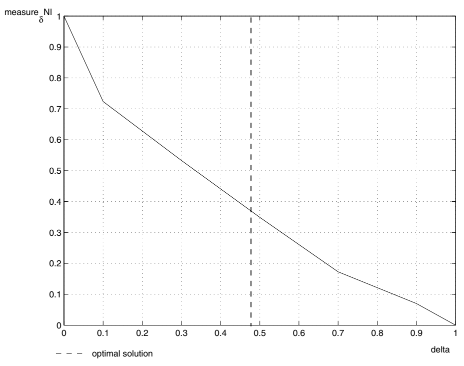
$measure_\alpha NI$ measures the "occurrence" of NI $_\alpha$interchangeable value pairs in the sense that it computes the average number of $_\alpha NI$ interchangeable pairs values over the whole CSP divided by the potential number of relations using the formula:

$$measure_{\alpha}NI = \frac{\sum_{k=1}^{n} \frac{\alpha NIV_k * 2}{domSize_{V_k} * (domSize_{V_k} - 1)}}{n}.$$

In the formula, n represents the problem size and $_{\alpha}NI\ V_k$ all the $_{\alpha}$interchangeable pairs values for variable V_k.



(a)



(b)

Fig. 3. A statistical measure of the number of NI of α/δ interchangeability for Fuzzy CSP with uniform weight distribution, for sets of problems with 10 variables and 10 values domains (Figure (a) for α and Figure (b) for δ).

Similarly, $measure_{\delta}NI$ measures the "occurrence" of NI $^{\delta}$interchangeable values:

$$measure_{\delta}NI = \frac{\sum_{k=1}^{n} \frac{\delta NIV_k * 2}{domSize_{V_k} * (domSize_{V_k} - 1)}}{n} -$$

As before, n represents the problem size and $^{\delta}NIV_k$ all the $^{\delta}$interchangeable pairs values for variable V_k.

In Figure 3(a), we represent $measure_{\alpha}NI$ varying the value of α. We found that there are a lot of $_{\alpha}$interchangeable values close to the optimal solution. We also notice that the interchangeabilities decrease when using lower values for α.

Similarly, in Figure 3(b), we observe also for δ the same results: *close to the optimum we have many* $^{\delta}/_{\alpha}$*substitutability/interchangeability.*

In the Figure 4, we represent how the occurrence of ($^{\delta}/_{\alpha}$)interchangeability depends on α and δ (respectively in Figure (a) and (b)), and also on the problem tightness.

As we can see in the Figure 4(a), the number of α interchangeable values depend on α, but also on the problem tightness. For low tightness, the number of interchangeabilities increases faster, while for higher values of tightness interchangeable values appear only for high values of α.

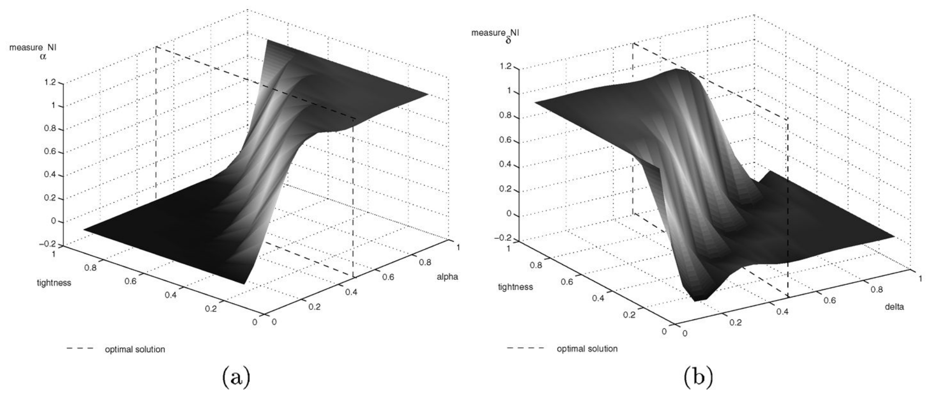


Fig. 4. A statistical measure of the number of α/δ interchangeability for Fuzzy CSP, with uniform weight distribution, for sets of problems with 10 variables and 10 values domains and varying the problem tightness.

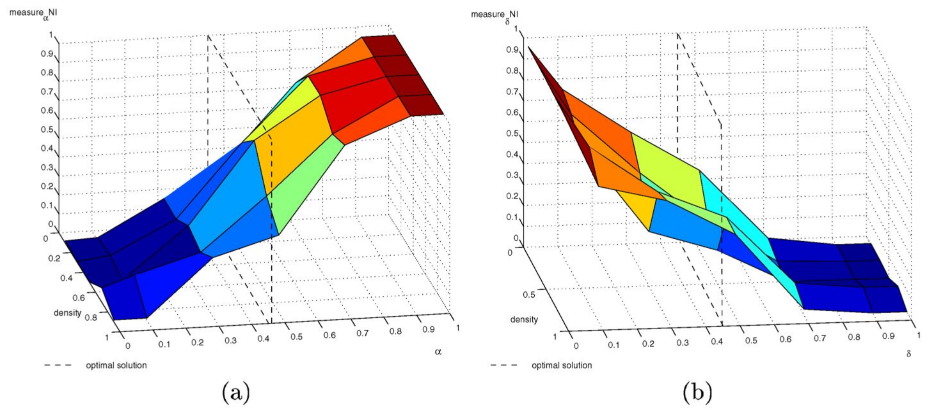


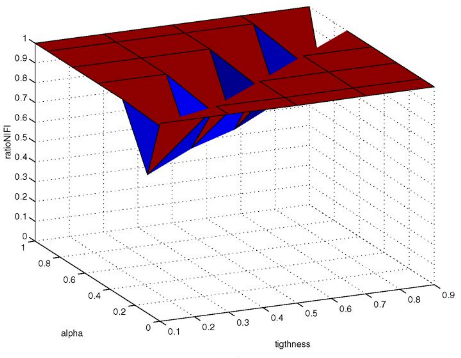
Fig. 5. A statistical measure of the number of α/δ interchangeability for Fuzzy CSPs with uniform weight distribution, for sets of problems with 10 variables and 10 values domains and varying problem density.

In Figure 4(b), we show the dependence w.r.t. δ and the problem tightness. We can see that the occurrence of interchangeable values increases with the tightness for low δ values.

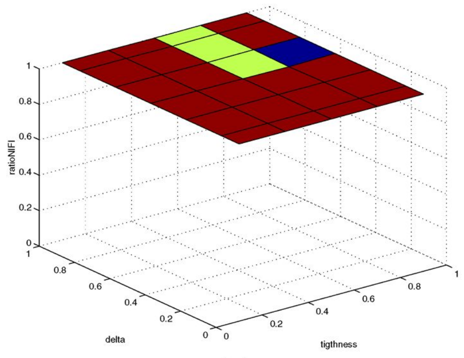
In Figure 5, we perform the same analysis but varying this time the density of the problem. We can notice that the interchangeability occurence does not vary too much with problem density (the shape is in fact very regular).

Estimation of NI versus FI for Fuzzy CSPs

Computing full interchangeable values might be a quite costly operation as it may require computing all the solutions. There are not known efficient algorithms



(a)



(b)

Fig. 6. The ratio between the number of neighbourhood and full interchangeable values varying tightness and α/δ (respectively on the left and right side of the picture), for problems of 10 variables with 10 domain values.

which can compute in polynomial time full interchangeable values. Neighbourhood interchangeability can be computed in polynomial time, but provide only a subset of full interchangeable values. In the following we study how neighbourhood interchangeability can approximate full interchangeability in soft CSPs.

We consider in Figure 6 the ratio between the number of neighbourhood interchangeabilities and the number of full interchangeabilities. The value of $ratioNIFI$ is computed as

$$ratioNIFI = \frac{\sum_{k=1}^{n} \delta NIV_k}{\sum_{k=1}^{n} \delta FIV_k}$$

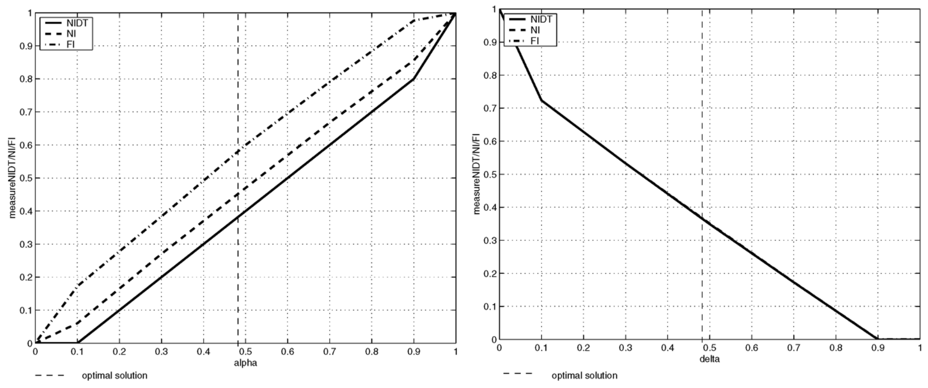
for δ interchangeability, and in a similar manner also for α. In the formula, ${}^{\delta}NIV_k$ represents the number of ${}^{\delta}NI$ interchangeable values pairs for variable V_k and ${}^{\delta}FIV_k$ represents the number of ${}^{\delta}FI$ interchangeable values pairs for variable V_k.

In Figure 6, we see that the ratios ${}^{\delta}NI/{}^{\delta}FI$ and ${}_{\alpha}NI/{}_{\alpha}FI$ are always between 0.7 and 0.9 when we consider domain values close to the optimal solution. Thus, NI interchangeability can well approximate FI interchangeability.

Estimation of NI Computed by General Definition versus NI Computed with the Discrimination Tree algorithm for Fuzzy CSPs

For soft CSPs based on idempotent semiring as Fuzzy CSP, we can use the proposed Discrimination Tree algorithm for computing $({}^{\delta}/{}_{\alpha})$neighbourhood interchangeability, see Section 2. In this paragraph we study how much the number of interchangeability values found with Discrimination Tree algorithm can approximate the number of interchangeability values detected with the definition algorithm and full interchangeability values respectively.

In Figure 7(a) we can see how ${}_{\alpha}$ full interchangeability, ${}_{\alpha}$ neighbourhood interchangeability computed using the definition algorithm and discrimination



(a) (b)

Fig. 7. How the number of α NI, α FI, α NIDT values are varying with α and δ.

tree algorithm respectively vary with α. The results show that the Discrimination Tree general algorithm finds a high number of interchangeable values.

Figure 7(b) represents the analysis for $^{\delta}$interchangeability. We can see that the graphs almost overlaps. Thus the number of interchangeability values found for $^{\delta}$neighbourhood/full interchangeability is almost the same. Thus, in the δ case the number of interchangeable values found by the Discrimination Tree algorithm is very close or almost the same to the original number found by the definition algorithm.

Following these results we can get to the conclusion that full interchangeability in Fuzzy CSPs, and thus for any idempotent soft CSP, can be well approximated by neighbourhood interchangeability computed either with the definition algorithm or with the Discrimination Tree algorithm.

4.2 Weighted CSPs

Weighted CSP represents the important class of Soft CSPs dealing with a non-idempotent times operator. In the following, we evaluate how the neighbourhood interchangeability can approximate full interchangeability.

δNI Occurrence in Weighted CSPs

We present how the occurrence of $^{\delta}$ neighbourhood interchangeable values varies with δ. The tests are not done for $_{\alpha}NI$ because when times is not idempotent we cannot easily compare local and global notions of α interchageabilities.

In Figure 8(a), we see how the number of $^{\delta}$neighbourhood interchangeable increases with δ and how, close to the optimal solution approaches to 1. This means that all the values pairs are interchangeable for high δ (as could be easily guessed).

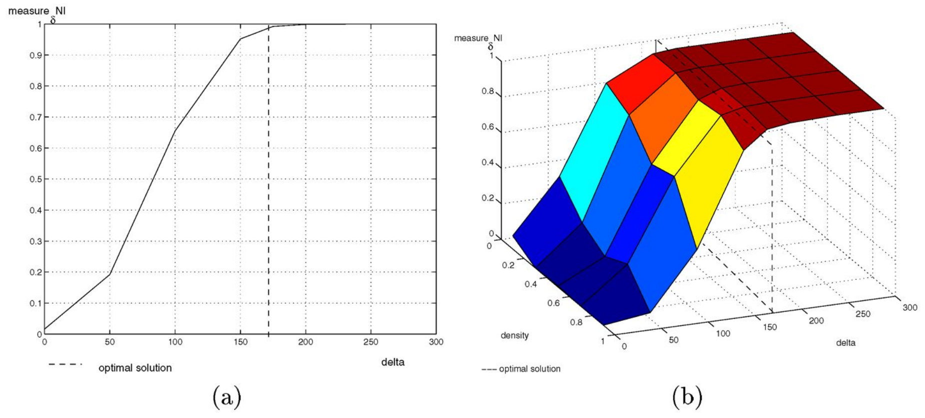


Fig. 8. How the number of interchageabilities varies with δ and with CSP density in weighted CSPs.

In Figure 8(b), we represent how the measure of NI varies w.r.t. δ and the density as well. We can see, as in the Fuzzy case, that the SCSP density does not influence the occurrence of interchangeability too much.

Estimation of NI versus FI for Weighted CSPs

In Figure 9, we can see how the number of neighbourhood and full interchangeability values vary with δ. We can see that the number of neighbourhood and full interchangeability does not differ too much. Thus, we can again approximate full interchangeability with neighbourhood interchangeability.

As in the tests for fuzzy CSPs, we computed the ratio $ratioNIFI$, and in Figure 10 we display the obtained results. We can see how the ratio between $^{\delta}NI$ and $^{\delta}FI$ varies with delta and CSP density. The ratio is always between 0.8 and 1 and this lead us to the conclusion that neighbourhood interchangeability can approximate fairly full interchangeability for weighted CSP if we can accept a degradation of δ.

5 Conclusions and Future Work

Interchangeability is an important technique as it can be used as a preprocessing technique for finding equivalences among variables values, and thus enhancing search. It can also be used as a postprocessing technique for solution adaptation. In this paper we used our java based implementation to compute interchangeability for Soft CSPs. We have studied the occurrence of $_{\alpha}NI$ and $^{\delta}NI$ and we have evaluated how this occurrence can depend on the values of α and δ, on the CSP parameters and also how local NI relies with FI. The experimental facts show also that there is high occurrence of $_{\alpha}NI$ and $^{\delta}NI$ interchangeability close to the optimal solutions in fuzzy CSPs and weighted CSPs as well. Thus, these algorithms can be used successfully in solution update applications. On random

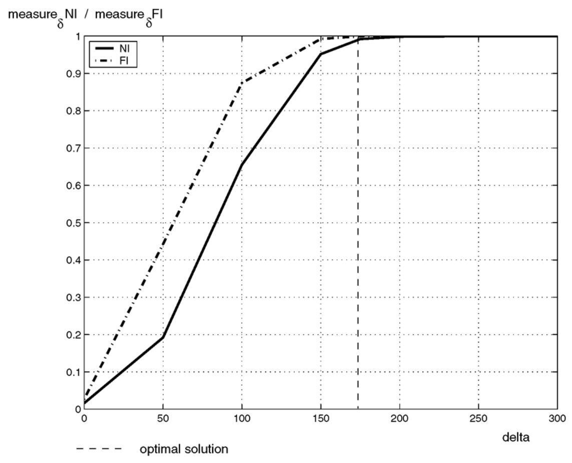


Fig. 9. δNI versus δFI values varying δ for weighted CSPs.

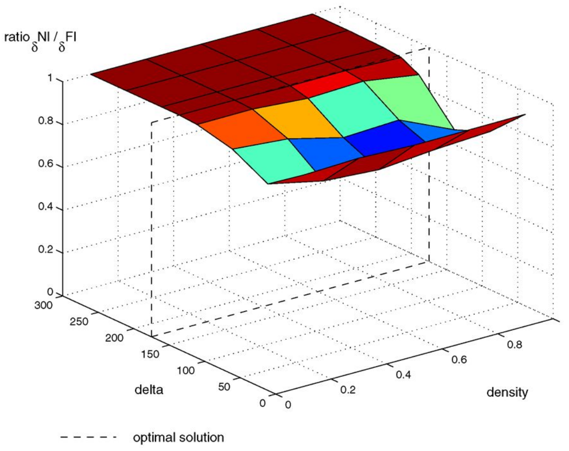


Fig. 10. The figure represents how ratio δNI/δFI is varying with δ and CSP density for weighted CSPs.

generated Soft CSPs we haved studied the occurence of ${}^{\delta}_{\alpha}$interchangeability varying with the CSP parameters such as size, density, tightness, and semiring values, where we covered all the range of Soft CSPs problems from easy to difficult.

We have noticed that there is high occurrence of interchangeability values by varying these parameters too. This motivates for the application of these methods in practical soft CSPs as a purpose of our further study.

Moreover, we showed that *NI* interchangeability can well approximate *FI* interchangeability. We studied also how the discrimination tree algorithm can be used to compute a subset of interchangeabilities.

The use of these interchangeability techniques in improving search have to be investigated. We believe that the results prove the reliability for using ${}^{\delta}_{\alpha}$interchangeability for solution updating and motivate for further studying.

Most work in constraint satisfaction has focused on efficielty generation solutions. However, it is also interesting how solutions can be adapted and abstracted. Interchangeability is an interesting concept for this. We believe that a computable form of interchangeability, as *neighbourhood interchangeability* is proposed in this paper, can be successfully applied for abstraction, for improving search and for solution updating in Soft CSPs.

References

1. B.W. Benson and E. Freuder. Interchangeability preprocessing can improve forward checking search. In *Proc. of the 10th ECAI*, 1992.
2. S. Bistarelli, B. Faltings, and N. Neagu. A definition of interchangeability for soft csps. In *Proc. ERCIM/CologNet Workshop on Constraint - Selected Papers*, LNAI. Springer-Verlag, 2002. to appear.
3. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2), 1997.
4. E. Bruchez and M. Torrens. Java constraint library. http://liawww.epfl.ch/ torrens/Project/JCL/, 1996.
5. B. Choueiry, B. Faltings, and R. Weigel. Abstraction by Interchangeability in Resource Allocation. In *Proc. of the 14th IJCAI-95*, pages 1694–1701, Montreal, Canada, 1995.
6. B.Y. Choueiry. *Abstraction Methods for Resource Allocation.* PhD thesis, EPFL PhD Thesis no 1292, 1994.
7. D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*, 1993.
8. E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proc. of AAAI-91*, 1991.
9. E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence Journal*, 58, 1992.
10. A. Haselbock. Exploiting interchangeabilities in constraint satisfaction problems. In *Proc. of the 13th IJCAI*, 1993.
11. N. Neagu, S. Bistarelli, and B. Faltings. On the computation of local interchangeability in soft constraint satisfaction problems. In *Proc. of the 16th International FLAIRS Conference - Special Track on Constraint Solving and Programming*, St. Augustine, Florida, USA, 2003. AAAI Press.
12. Nicoleta Neagu and Boi Faltings. Constraint Satisfaction for Case Adaptation. In *In Proc. of the workshop sesion (ICCBR99)*, 1999.
13. Zs. Ruttkay. Fuzzy constraint satisfaction. In *Proc. 3rd IEEE International Conference on Fuzzy Systems*, 1994.
14. T. Schiex, H. Fargier, and G. Verfaille. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proc. IJCAI95*, 1995.
15. R. Weigel and B. Faltings. Interchangeability for case adaptation in configuration problems. In *Proc. of the AAAI98 Spring Symposium on Multimodal Reasoning, Stanford, CA*, 1998. TR SS-98-04.
16. R. Weigel and B. Faltings. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115, 1999.

A Rule Language for Interaction[⋆]

Carlos Castro[1], Eric Monfroy[2], and Christophe Ringeissen[3]

[1] Universidad Técnica Federico Santa María, Valparaíso, Chile
Carlos.Castro@inf.utfsm.cl
[2] IRIN, Université de Nantes, France
Eric.Monfroy@irin.univ-nantes.fr
[3] LORIA – INRIA-Lorraine Villers-lès-Nancy, France
Christophe.Ringeissen@loria.fr

Abstract. In this paper, we propose a rule language to design interactive component languages and basic coordination languages. In our language, concurrent rules manage interactions of call-back functions that apply on a store of data. This store can freely be structured as an array, a list, a set of communication channels, etc. Our rule language can be seen as an abstract machine to specify and implement interactive component languages. We also propose such a component language devoted to solver cooperations and solver cooperation languages. We illustrate the use of this specific component language to implement some primitives of an existing solver cooperation language.

1 Introduction

The rule-based programming paradigm was introduced in the early seventies and more recently there has been a revival interest in this approach. As an example of these new developments we can mention the ELAN language [5] which is based on term rewriting: a program consists of a set of conditional rewrite rules which are controlled by strategies to specify their order of application. Another example is the CHR language [8] which extends the syntax of a host language (such as ECL^{i}PS^e, Sicstus, or Java) with rules for rewriting constraints.

Solver cooperation is a research topic that has been investigated during the last years by the Constraint Programming community [9,15,4,2]. Considering that very efficient constraint solvers are currently available, the challenge is to make them cooperate in order to 1) solve hybrid problems that cannot be treated by a single solver, 2) solve distributed problems (either naturally distributed or for some security and confidentiality reasons), 3) improve solving efficiency, and/or 4) reuse (parts of) solvers to reduce implementation costs. Solver cooperation languages (such as the ones proposed in [13,7,10]) provide one with primitives to manage cooperation. However, implementing such languages is a

[⋆] The authors have been partially supported by the Franco-Chilean INRIA-CONICYT project Cocars, by the European IST project COCONUT, and by the Chilean National Science Fund through the project FONDECYT 1010121.

tedious task since instructions are complex and do not make a clear separation between computation, strategies, and interaction/coordination.

More recently, a component-based framework has been proposed for designing and implementing constraint solver cooperations and solver cooperation languages [12]. Combining these basic components into patterns enables one to manage computation, control, and coordination required for solver cooperations. The interactive behavior of these components is described by concurrent rules.

In [12], the main motivation is to provide a simple and basic component language for describing solvers involved in cooperations, and for controlling their interactions. In this paper, we extend the ideas of [12] and we propose a rule language for designing and implementing such interactive component languages and basic coordination languages [14].

In our language, rules manage interactions of call-back functions that apply to a store of data. This store can freely be structured as an array, a list, a set of communication channels, etc. Our rules apply concurrently, and informally their behavior is the following: if it can be checked that some data (possibly schematized by variables) are present at some given addresses of the store, then apply some call-back functions to these data and store the results at some given addresses in the store. For example, consider that addresses of the store are ports connecting channels, and consider the following rule:

$$chk(i, X) \Rightarrow put(o_1, f(X)), put(o_2, g(X))$$

Then, as soon as (and as long as) some data is present on port i, the rule fires: the data is stored in the local variable X; the data is removed from port i; functions f and g are applied to X and the result is stored (i.e., sent when using channels) on some output ports ($f(X)$ is stored on port o_1 and $g(X)$ on port o_2). Note that f and g can be constraint solvers.

These rules can then be combined into components to provide component languages for interaction. These components have a more complex behavior than a single rule and can be seen as some wrappers providing coordination and interactive features to functions. Consider again that addresses are ports, and consider the following component made of two rules:

$$\left\{ \begin{array}{ll} chk(i_1, X), chk(i_2, true) & \Rightarrow put(o_1, f(X)), \\ chk(i_1, X), chk(i_2, false) & \Rightarrow put(o_2, g(X)) \end{array} \right\}$$

As soon as some data is present on port i_1, and that the data $true$ or $false$ is present on i_2, one of the two rules fires. If $true$ is present on i_2, then f is applied to the data on i_1 (this data is then removed), and the result is sent on o_1. If $false$ is present on i_2, the process is similar, but this time $g(X)$ is sent on o_2. If there is no data on i_1, or no data (or data different from $true$ or $false$) on i_2, then nothing happens. Indeed, these two rules also synchronize "messages" sent on the two ports i_1 and i_2. We can see the interest of such a component for solver cooperation: depending on a property (received on port i_2) checked by a solver (working somewhere else at the same time) a solver f or a solver g

is applied to some data (received on port i_1) and forwarded either to port o_1 or port o_2.

Moreover, rules and components can be parameterized by addresses, data, and functions: thus, they define a single behavior, but their computations, inputs, and outputs can vary with respect to the parameters.

In order to ease programming and to enable code re-use, one can write some complex strategies in terms of patterns using a *component language* (i.e., a set of components). A pattern is a combination of components and patterns that can be parameterized by data, addresses, and functions. One can thus incrementally build some patterns having more and more complex behaviors and managing more and more functions. Compared to standard languages, our rule language can be seen as an abstract machine, component languages as coordination-based programming languages (a component being an instruction), and patterns as procedures.

We used our rule language to improve and to extend the component language proposed in [12]: components are now better and more formally defined, they do not need initialization by hand, and the initialization of the data-flow is now performed by components. This component language has been used to implement some of the primitives of the language for constraint solver cooperation of [7].

The paper is organized as follows: we introduce in Section 2 our notion of stores, and the operations allowed on this structure. In Section 3, we present the rules we are interested in. Section 4, we consider a rule-based language where rules are first encapsulated into components, and where components are then reused to build more complex reusable patterns. We instantiate this framework in Section 5 by presenting some particular components and patterns, and we show their interest for a solver cooperation problem. We conclude in Section 6.

2 Basic Definitions and Notation

In this section, we introduce basic definitions and notations used in this work. First we define the notions of data and store and then we present the only three operations that apply to the store.

Data and Stores. We consider $\mathcal{D}$ a set of data[1] containing the *nil* object. A store is a set of cells to collect data that are accessed concurrently: data can have several occurrences in a store, and some cells can be empty. Cells are identified by a unique address that is used to access and modify data. In the following, we abstract the notion of store by a set of addresses $\mathcal{A}$.

[1] Data of $\mathcal{D}$ can be of several types. However, in this paper we will neglect this aspect although this is of great interest especially when linking components and patterns.

Example 1 (Stores). Stores are very diverse, and can be for example:

– an array, addresses being indices,
– a list, addresses being positions in the list,
– a set of ports for communication channels where messages are buffered in a FIFO way on ports (extremity of a channel). This time, the address is composed of the number/name of the port, and the position in the FIFO buffer. However, as we just want to access the last element of a FIFO, the name of the port is sufficient to get/check/read the message (position is implicit since it is the last element of the buffer). Similarly, the name of the port is sufficient for sending messages (put operator): implicitly, the address is completed by considering the first position of the FIFO.

Operators on Stores. In the following, we consider a given set of addresses $\mathcal{A}$, and a given set of data $\mathcal{D}$.

Since stores are concurrently consulted and modified, we associate each address with a binary semaphore to perform mutual exclusion access to data. A binary semaphore is a shared variable (value 0 or 1) that supports two operations: P and V. If a P operation is made on a semaphore s and $s = 1$, then s is set to 0. If $s = 0$, then the P operation blocks until $s = 1$. A V operation on a semaphore s never blocks and sets s to 1. In the following, we denote the semaphore associated to an address by the name of the address itself. Moreover, we denote by $\mathsf{P}(a_1, \ldots, a_l)$ or $\mathsf{P}_{i\in[1..l]}(a_i)$ the simultaneous access request to $a_1, \ldots, a_l$. The P operation blocks until every semaphore associated to $a_1, \ldots, a_l$ is set to 1: if there exists at least one a_i set to 0, none of the remaining a_j is locked by this P operation. This avoids inter blocking configuration that could happen by performing sequences of P, e.g., performing $\mathsf{P}(a), \mathsf{P}(b)$ and $\mathsf{P}(b), \mathsf{P}(a)$ concurrently; the first one getting the resource a, and the second one getting the resource a. Note that each semaphore is initialized to 1.

We consider three basic operators for accessing/modifying data of a store $\mathcal{A}$ [2]:

– *chk* to test existence of data in the store,
– *put* to save data,
– *delete* to remove data.

chk operator. Let us introduce a set of data variables, denoted by $\mathcal{V}_{\mathcal{D}}$. The syntax of the *chk* operator is $chk(a, d)$ where $a \in \mathcal{A}$ and $d \in \mathcal{D} \cup \mathcal{V}_{\mathcal{D}}$. The behavior of $chk(a, d)$ is defined as follows:

– $chk(a, X)$ with $X \in \mathcal{V}_{\mathcal{D}}$ succeeds if some data d (different from the *nil* object) is stored at a. In this case, X is instantiated by d,
– $chk(a, d)$ with $d \in \mathcal{D}$ succeeds if d is stored at a,
– otherwise, $chk(a, d)$ fails.

[2] Operator definitions should be overloaded by the name of the store. However, we consider a unique store, and thus we do not need this extra information.

put operator. The syntax of the *put* operator is $put(a, d)$ where $a \in \mathcal{A}$ and $d \in \mathcal{D}$. A *put(a,e)* operation stores d at the address a. One can remark that the data d is possibly obtained by evaluating a complex data expression, provided that we have functions on data and data variables which are assigned by *chk* operations.

del operator. The syntax of the *del* operator is $del(a)$ where $a \in \mathcal{A}$. A *del(a)* operation deletes data from the address a (i.e., stores the *nil* object at a, so $del(a)$ is equivalent to $put(a, nil)$).

Note that operators applying to the same address must be executed in mutual exclusion using semaphores. We also consider compound operators to apply several operators simultaneously.

Example 2 (Operators for communication channels). Consider the store is a set of uniquely identified ports connected by directional channels for exchanging messages. Assume that messages are read in the same order they are produced, and thus, messages are buffered on ports in a FIFO structure. Then, a possible instance of the $chk(a, d)$ operator is to consider that a is the name of the port, d the message we are looking for, and that $chk(a, d)$ succeeds when d is the first message pending on a, i.e., the oldest not yet-consumed message sent to a through the channel. $del(a)$ consists in consuming the oldest message stored on port a, and $put(a, d)$ consists in sending d on the channel connected to a: d becomes the newest message stored on a.

3 The Rule Language

In this section, we formally describe the notion of rules we use in our language and their operational semantics.

Consider a set of addresses $\mathcal{A}$, a set of data $\mathcal{D}$, and a set of functions $\mathcal{F}$ over $\mathcal{D}$. In addition, we consider a set of address variables $\mathcal{V}_{\mathcal{A}}$, a set of function variables $\mathcal{V}_{\mathcal{F}}$, and a set of data variables $\mathcal{V}_{\mathcal{D}}$. These variables are used to define generic expressions. They are instantiated at different levels in our framework. Address and function variables are used to express generic rules. Instantiating address and function variables in generic rules leads to executable "closed" rules. Note that the only data variables that still occur in closed rules, are instantiated by *chk* operations when rules are fired. A generic rule cannot be fired since we cannot know where to check data, where to put data, and which function to apply. However, generic rules are very important since they describe a behavior (in terms of interaction or coordination) independently from the computation (given by function parameters) and the addresses used to get and put data (given by address variables). Thus, several closed rules can be obtained by different instantiations of a single generic rule: these closed rules have the same behavior, but describe different computations, and use different addresses to get and put data.

3.1 Expressions

A $\mathcal{F}$-expression e is either:

- a data variable $v \in \mathcal{V}_{\mathcal{D}}$,
- a data $d \in \mathcal{D}$,
- or a term $f(d_1, \ldots, d_l)$ where f is a function symbol of arity l in $\mathcal{F}$ and for all $i \in [1..l]$, $d_i \in \mathcal{D} \cup \mathcal{V}_{\mathcal{D}}$.

3.2 Rules

Definition 1 (Rule). *Let $A \subseteq \mathcal{V}_{\mathcal{A}}$ and $F \subseteq \mathcal{V}_{\mathcal{F}}$ be, respectively, a set of address parameters and a set of function parameters. A rule parameterized by (A, F) is a pair $l \Rightarrow r$ such that*

- *the left-hand side l is $\bigcup_{i \in I} chk(a_i, d_i)$ where*
  - $\forall i \in I, a_i \in \mathcal{A} \cup A$
  - $\forall i \in I, d_i \in \mathcal{D} \cup \mathcal{V}_{\mathcal{D}}$
  - $\forall i, i' \in I, i \neq i' \Rightarrow a_i \neq a_{i'}$
- *the right-hand side r is $\bigcup_{j \in J} put(a_j, e_j)$, where*
  - $\forall j \in J, a_j \in \mathcal{A} \cup A$
  - $\forall j \in J$, *e_j is a $(\mathcal{F} \cup F)$-expression*
  - $\forall j, j' \in J, j \neq j' \Rightarrow a_j \neq a_{j'}$
- *The set of data variables occurring in r is included in the set of data variables occurring in l.*

When A and F are empty, the rule is said to be closed. The set of closed rules is denoted by $\mathcal{R}$.

Since the way of instantiating address and function variables is completely standard, we just illustrate some possible instantiations of a given parameterized rule.

Example 3 (Rule instances). The following rule

$$chk(V_A, x) \Rightarrow put(V_B, V_F(x)), put(V_A, V_G(x))$$

is parameterized by $(\{V_A, V_B\}, \{V_F, V_G\})$, where x is a data variable which is instantiated when the rule is fired. Instantiating V_b by $b \in \mathcal{A}$ and V_F by $f \in \mathcal{F}$, we get the rule instance

$$chk(V_A, x) \Rightarrow put(b, f(x)), put(V_A, V_G(x))$$

which is still a parameterized rule. Instantiating the latter as follows: V_A by $a \in \mathcal{A}$ and V_G by $g \in \mathcal{F}$, we get:

$$chk(a, x) \Rightarrow put(b, f(x)), put(a, g(x))$$

which is a closed rule. It is important to remark that a closed rule can contain data variables.

3.3 Rule Applications

In our framework we consider that only closed rules can be executed. The application of the rules can be carried out either each time they are applicable or just the first time they are applicable. We express this control using the *Apply Always* and *Apply Once* operators.

Apply Always The operational semantics of the forever application of a rule r

$$\bigcup_{i \in I} chk(a_i, d_i) \Rightarrow \bigcup_{j \in J} put(a_j, e_j)$$

is defined as follows:

```
apply($r$):
    $\mathsf{P}_{i \in I \cup J}(a_i)$
    if $chk(a_i, d_i)$ succeeds $\forall i \in I$
      then  % the head succeeded
            $del(a_i) \forall i \in I,$
            $\mathsf{V}_{i \in I \setminus J}(a_i)$
            $put(a_j, e_j)\} \forall j \in J$
            $\mathsf{V}_{j \in J}(a_j)$
      else  % the head failed
            $\mathsf{V}_{i \in I \cup J}(a_i)$
    endif
    apply($r$)
```

Note that apply(r) is a non-terminating process, where one always tries to apply r.

Apply Once This operator consists in applying a rule only the first time it applies. The operational semantics of a single application of a rule r is obtained by removing the recursive call apply(r) from the above definition and by adding the same kind of recursive call in the "else" part:

```
applyS($r$):
    $\mathsf{P}_{i \in I \cup J}(a_i)$
    if $chk(a_i, d_i)$ succeeds $\forall i \in I$
      then  % the head succeeded
            $del(a_i) \forall i \in I,$
            $\mathsf{V}_{i \in I \setminus J}(a_i)$
            $put(a_j, e_j)\} \forall j \in J$
            $\mathsf{V}_{j \in J}(a_j)$
      else  % the head failed
            $\mathsf{V}_{i \in I \cup J}(a_i)$
            applyS($r$)
    endif
```

Notation: by convention, we use different notations for rules depending on the application strategy: a rule applied *always* is denoted as usual with the arrow ($\Rightarrow$), whereas a rule applied *once* is denoted by ($\rightarrow$).

We say that a rule fires as soon as its left-hand side succeeds. In this case, variables appearing in check operators are instantiated with data found at the given addresses. Data are then deleted before applying functions and storing new data.

Remember that no uninstantiated variable remains after the rule fires (each variable in a put operation also appears in a check operation) and thus, the *put* operations occurring in the right-hand side of the rule is well-formed.

Firing a rule is a non interruptible process due to the mutual exclusion mechanism that blocks all addresses used by the rule (in check and put). When the left-hand side succeeds, delete and put operations are executed and cannot be stopped. During this time, no other rule can access the addresses needed by a fired rule.

A *program* is a set of concurrent closed rules. Single application rules are fired at most once: once if the head succeeds (in this case, applyS(r) terminates), no time if the head never succeeds (applyS(r) does not terminate), whereas forever application rules are fired as many times as their head succeeds. The store being non "monotonic", a rule head can fail and later succeed.

4 Languages

The rule language defined in the previous section is now used to write component languages: these languages are sets of instructions, each instruction being a union of rules (a *component*).

The notion of patterns (sets of components and patterns) helps programming in these languages. Patterns can be seen as some procedures in terms of imperative programming, or patterns of interaction in terms of coordination models. Instances of patterns (i.e., partial instantiations) ease the design and re-use of patterns and enable one to incrementally built complex reactive programs from basic bricks.

For execution, patterns and components are "flattened" into a set of rules that execute concurrently.

4.1 Components

Our rule language is now used to write languages based on components. A component is defined by the union of a set of rules allowing us to implement instructions. The parameters of a component are built from the parameters of the composing rules.

Definition 2 (Component). *Given a set of address variables A, a set of function variables F, and a component identifier co, the parameterized component* *denoted by $co(A,F)$ is a set of rule instances obtained by replacing their parameters with elements respectively in $\mathcal{A} \cup A$ and in $\mathcal{F} \cup F$. A* component language *is a set of parameterized components, each of them written with a given set of parameterized rules.*

4.2 Patterns

A program written with a component language as defined above is limited to a set of components. To ease programming and to enable code re-use, we propose the notion of patterns. A pattern is a parameterized piece of program written as a set of components and patterns.

Pattern instances are partial instantiations of patterns: they can be used as programs or as pieces of programs to incrementally built more complex programs.

Definition 3 (Patterns). *Given a set of address variables A and a set of function variables F, the set of patterns parameterized by (A,F), denoted by $\mathcal{P}(A,F)$, is inductively defined as follows: given a pattern identifier p, the* pattern $p(A,F)$ *is either*

- *a component instance,*
- *or a set of pattern instances,*

obtained by replacing their parameters with elements respectively in $\mathcal{A} \cup A$ and in $\mathcal{F} \cup F$. When A and F are empty, the pattern is said closed. The set of closed patterns is denoted by $\mathcal{P}$.

4.3 Execution of Programs

The execution model of our languages is very simple. Closed patterns are *flattened* recursively until a set of concurrent closed rules is obtained. Since only closed rules can be executed, only closed components and closed patterns can be executed. The *flatten* function is inductively defined from $\mathcal{P}$ to $\mathbf{2}^{\mathcal{R}}$ (the powerset of $\mathcal{R}$):

- $flatten(\bigcup_{i \in I}\{p_i\}) = \bigcup_{i \in I} flatten(p_i)$ if p_i's are pattern instances in $\mathcal{P}$,
- $flatten(\bigcup_{i \in I}\{r_i\}) = \bigcup_{i \in I}\{r_i\}$ if r_i's are rule instances in $\mathcal{R}$.

The execution of a closed pattern p corresponds to the concurrent execution of rules in $flatten(p)$.

5 A Component Language

We now use our rule language to design a specific component language to manage interaction of processes (in the sense of Manifold [3], i.e., an abstraction for system processes, threads, ...). The store we consider is a set of FIFO buffers for ports connected by one-to-one directional communication channels (defined in the last item of Example 1), and the store operators are the ones defined in Example 2:

– *chk* is a read that does not consume messages on channels,
– *del* consumes a message on a channel,
– *put* sends messages to channels.

This language is then used to write interaction patterns (Section 5.2), and finally these patterns are used to write a solver cooperation language (Section 5.3).

Note that we defined the store and the operators whereas the component language could be described with a generic store. This is for sake of clarity: the reader can have a better view of the language for managing interaction, and its application to coordinate solver cooperation. For example, by considering a store structured as a list of FIFO, we believe that we can use the same component language to describe the strategy language of ELAN.

5.1 The Language

The language we now present is an improvement of the one of [12]: an *init* component has been added, the $y - junction$ component clarifies channel connections, and a channel is now a component named *connect*.

Connectors. A *connect* component represents a directional channel linking two ports O and I in a directional way. A message stored on the FIFO buffer of an output port O is forwarded to the FIFO buffer of the input port I. This component consists of only one rule:

$$connect([O, I], []) = \{chk(O, X) \Rightarrow put(I, X)\}$$

Y-Junctions. A *Y_junction* component links two input ports to an output port: each time there is a message on either I_1 or I_2 it is forwarded to the output port O. This component is written as:

$$y\_junction([I_1, I_2, O], []) = \left\{ \begin{array}{l} chk(I_1, X) \Rightarrow put(O, X) \\ chk(I_2, X) \Rightarrow put(O, X) \end{array} \right\}$$

Initializers. An *init* component puts a data (the value of a constant function) on an output port. This type of components is meant for initializing data flow, i.e., at the beginning, all channels are empty (the store is empty), and thus, no rule (or component) can be fired.

$$init([O], [X]) = \{\to put(O, X)\}$$

Transformers. Components of this type have one input port I, one output port O, and a parameter function F. When they receive a message X on I, they consume it, apply F on X, and send the result on O.

$$\mathsf{trans}([I, O], [F]) = \{chk(I, X) \Rightarrow put(O, F(X))\}$$

Note that a *connect* component is the instance $trans([I, O], [id])$ where id is the identity function of a *transformer* component. However, for some clarity reasons, we consider both types of components.

Synchronizers. These components synchronize two components by the mediating of their messages. They wait for two messages (one on the input port I_1 and one on the input port I_2) before applying the function parameter F on the two messages and putting the result on the output port O:

$$\mathsf{sync}([I_1, I_2, O], [F]) = \{chk(I_1, X), chk(I_2, Y) \Rightarrow put(O, F(X, Y))\}$$

First. A first component has four input ports (I_1, I_2, ii_1, and ii_2) and one output port O. Ports ii_1 and ii_2 are used as local memories: a first component can put (store) and check (recall) some messages on them. Ports ii_1 and ii_2 are not accessible from outside of the component, and are not connected to any channel. Thus, they do not appear in the list of variables of the component.

For a couple of messages, one arriving on I_1 and the other one on I_2, only the first one arrived is immediately put on O, the other one will be destroyed when it will arrive:

$$\mathsf{first}([I_1, I_2, O], []) = \left\{ \begin{array}{rl} & \rightarrow chk(ii_1, false), \\ & \rightarrow chk(ii_2, false), \\ chk(I_1, d), chk(ii_2, true) & \Rightarrow put(ii_2, false), \\ chk(I_2, X), chk(ii_1, true) & \Rightarrow put(ii_1, false), \\ chk(I_1, X), chk(ii_2, false) & \Rightarrow put(ii_1, true), put(O, X) \\ chk(I_2, X), chk(ii_1, false) & \Rightarrow put(ii_2, true), put(O, X) \end{array} \right\}$$

When *true* is put on ii_1, this indicates that the *first* component has already read a message on I_1, and forwarded it to O. If the message on ii_1 is $false$, either no message has been read yet on I_1, or the component has been reset. The reset is performed by the 3rd rule (respectively the 4th rule) which destroys the message on I_1 (respectively I_2), and reinitialize the component by putting $false$ on ii_2 (respectively ii_1). This mechanism "cleans" the input ports of a first component to prepare for its later use with a new couple of messages. The first two single application rules are used only once to initialize the component.

Note that this initialization could also be performed using some *init* components linked to ii_1 and ii_2 that would become parameters and accessible from outside of the component.

Sieve. A sieve component waits for a message X on I_1, and for the message *true* or $false$ on I_2. If *true* is checked on I_2, then X is put on O; if $false$ is checked on I_2, then X is consumed but no action is performed (data X thus vanishes); otherwise (another value is checked on I_2) a *sieve* component is blocked forever[3].

$$\mathsf{sieve}([I_1, I_2, O], []) = \left\{ \begin{array}{ll} chk(I_1, X), chk(I_2, true) & \Rightarrow put(O, X), \\ chk(I_1, X), chk(I_2, false) & \Rightarrow \end{array} \right\}$$

Such a component blocks a message until a "control" arrives, and deletes it if the "control" is $false$. This avoids sending a message to a component C when we are not sure C will properly consume it, understand it, or really need it.

[3] This features focus on the need for type checking: here, data checked on I_2 should be boolean values to correctly use a *sieve* component.

Duplicate. A *duplicate* component duplicates a dataflow: it gets a message X on its input port I and returns a copy of X on both its output ports O_1 and O_2.

$$\mathsf{dup}([I, O_1, O_2], []) = \{chk(I, X) \Rightarrow put(O_1, X), put(O_2, X)\}$$

Note that except the *y_junction*, these components are deterministic: only one rule can be fired.

5.2 Some Patterns of Interaction

We now design some patterns of interaction that are similar to patterns in coordination models: a topology of agents is fixed, their connections and interactions are fixed, but the computational kernel of agents is generic; in our case, functions applied by agents are parameters of the patterns.

Implicit channels. Consider $a([\ldots, o, \ldots], \ldots)$ and $b([\ldots, i, \ldots], \ldots)$, two patterns such that o is connected to i inside a pattern c by a connect component: $c(\ldots) = \{a([\ldots, o, \ldots], \ldots),\ b([\ldots, i, \ldots], \ldots),\ \mathsf{connect}([o, i], []), \ldots\}$. When this does not cause any confusion, we can share a port between a and b by renaming i to o (or vice-versa). Thus, the connect component is removed, and the pattern is lightened: $c(\ldots) = \{a([\ldots, o, \ldots], \ldots),\ b([\ldots, o, \ldots], \ldots), \ldots\}$[4].

Multi-duplicates. A message must often be duplicated several times. We propose the multiDup_3 pattern to duplicate 3 times a message:

$$\mathsf{multiDup\_3}([I, O_1, O_2, O_3], []) = \{\mathsf{dup}([I, O_1, o], []), \mathsf{dup}([o, O_2, O_3], [])\}$$

and so on for each n, leading to the definition to duplicate n times a message:

$$\mathsf{multiDup\_n}([I, O_1, \ldots, O_n], []) = \left\{ \begin{array}{l} \mathsf{multiDup\_n\text{-}2}([I, O_1, \ldots, O_{n-2}, o], []), \\ \mathsf{dup}([o, O_{n-1}, O_n], []) \end{array} \right\}$$

Switch. A *switch* pattern receives an input m on port I and forward it to either O_1 or O_2. The decision is taken by an "external" pattern which receives m on Ip and returns a result (Boolean) on port Op. This is illustrated in Figure 1 where triangles are duplicators, lines are channels, and zig-zag lines are channels carrying Boolean values.

This switch pattern can be defined (and thus implemented) by (see Figure 1):

$$\mathsf{switch}([I, O_1, O_2, Ip, Op], []) = \left\{ \begin{array}{l} \mathsf{multiDup\_3}([I, s_1, Ip, s_2], []), \\ \mathsf{dup}([Op, sc_1, ni], []), \\ \mathsf{transf}([ni, sc_2], [not]), \\ \mathsf{sieve}([s_2, sc_2, O_2], []), \\ \mathsf{sieve}([s_1, sc_1, O_1], []) \end{array} \right\}$$

[4] This is similar to considering a store structured as channels (and not ports): agents write and read in channels instead of ports. This also means that there is no delay between sending a message in a channel and receiving it.

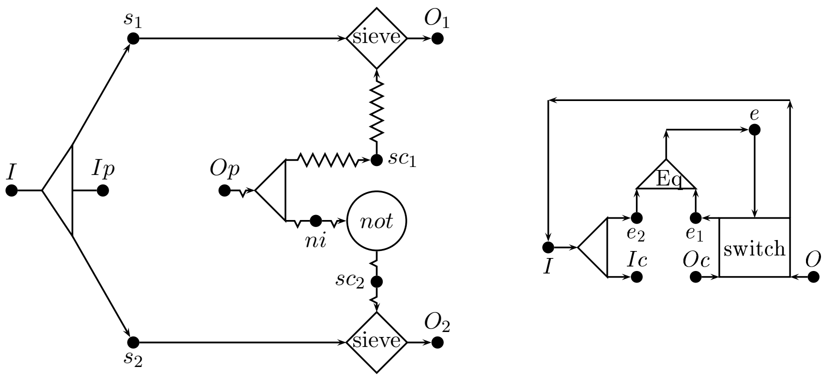


Fig. 1. Switch pattern and Fixed point pattern

Note that this switch pattern is more generic than the one presented in [12] which forces the pattern between ports Ip and Op to be a transformer calling a function to test whether a given property is true or not.

Fixed point. A fixed-point (see Figure 1) is a common instruction of solver cooperations: it applies iteratively a cooperation p (in our case, a pattern accepting Ic as input and Oc as output port) until the result of p does not change anymore, i.e., the input and the output of the cooperation are the same. The fixed-point pattern is defined as follows

$$\mathsf{fp}([I, O, Ic, Oc], [Eq]) = \left\{ \begin{array}{l} \mathsf{switch}([Oc, I, O, e_1, e], []), \\ \mathsf{sync}([e_1, e_2, e], [Eq]), \\ \mathsf{dup}([I, Ic, e_2], []) \end{array} \right\}$$

where I (respectively O) is the input (respectively output) port of the fixed-point, Ic and Oc are the input port and output port respectively of the pattern (cooperation) whose fixed-point we are looking for, and Eq is a function to test equality of two messages [5].

To illustrate the fp pattern, consider we want to compute the fixed point application of a function f. I is the address of the input of f, O the address where we want to store the fixed-point, $=$ the function to test equality of elements of the domain of f. Then, we just have to “encapsulate” f in a transformer component, and to link it to a fixed-point pattern. We obtain the pattern:

$$\mathsf{fp\_f}([I, O], []) = \{\mathsf{fp}([I, O, ic, oc], [=]), \mathsf{trans}([ic, oc], [f])\}$$

5.3 Application to Solver Cooperation

In this section, we present the implementation of some of the primitives of the cooperation language of [7,6]. We also present some executions of our components for constraint propagation, a standard method for constraint solving [1].

[5] The Eq function is a parameter since depending on the structure of messages, testing equality can be different and not only syntactic.

We first define the structure of problems (or messages) and the call-back functions we will use. We consider two types of atomic constraints over finite domain variables:

- $in(a, [lb_a, ub_a])$ defining the domain of $a \in [lb_a, \ldots, ub_a]$, and
- $leq(a, b)$ meaning $a \leq b$.

A conjunction is a list of atomic constraints; a disjunction is a list of conjunctions. We consider a solver *narrow_leq* which given a problem $leq(a, b)$, $in(a, [lb_a, ub_a])$, $in(b, [lb_b, ub_b])$ reduces the domain of a and b in order to make the problem locally consistent [11]. The filter function *filter_ldd* filters a *leq* constraint and the related domain constraints (i.e., a suitable input for the *narrow_leq* solver); *first* is a strategy that performs depth-first search (first element in the list with our constraint problem representation).

Basic primitives. The implementation of the basic primitives of [6] is straightforward with our component language:

- The sequence consists in applying a second solver cooperation on the result of a first solver cooperation. With our rules, the sequence consists of connecting the output of a pattern to the input of the next one.
- The don't care of [6] is a primitive that takes undeterministically the result of one solver cooperation. In our framework, it is a synchronizer component with a random function to choose any of the inputs ($dc(d_1, d_2) = d_1$ *or* $dc(d_1, d_2) = d_2$).
- The fixed-point primitive of [6] corresponds to our fixed-point pattern (see Section 4).

Best_apply. In [6], the semantics of this primitive is: given a problem p, make the best (w.r.t. some criteria given as parameters) possible application of a cooperation *coop* on a part (filtered using some criteria given as parameters) of p such that p is effectively modified. In other words, w.r.t. our framework: a problem implies several candidate problems; *coop* is applied on one of them only; and this candidate is the best one (w.r.t. some criteria) that *coop* will really modify.

The operational semantics of this primitive is rather complex. Thus, we divide the primitive into sub-primitives corresponding to patterns that are then grouped into a single pattern to obtain the operational semantics of the bestApply primitive.

We only describe some of the patterns. The csolve pattern takes as input a problem p, applies a pattern *coop* (the solver cooperation of bestApply) on p and has three output ports: if *coop* really modifies p, then *true* is put on the success port, p is put on the old port, and the result of applying *coop* is put on the new port. Otherwise, *false* is put on success, and nothing on old and new.

The cselect pattern creates a computational choice point (using a filter function). As long as it reads *true* on a more input port (that will be the negation of the success port of csolve), it sends candidates on the outC output port (that will be connected to the input of csolve). When $false$ is read on more, or when there is no more candidate (i.e., p is empty), cselect stops.

The declaration of the bestApply pattern (based on the problem structure described above, whereas a –structure independent– pattern is presented in [12]) is the following: bestApply$([In, Out, In_C, Out_C], [Filter, Selector])$, where In and Out are the input and output ports of the pattern. In_C and Out_C are the input and output ports of the cooperation pattern to apply. $Filter$ and $Selector$ are parameters to create and select the candidates: they represent the criteria given to the bestApply primitive). Here is an example of use of our pattern: $\{$bestApply$([i, o, ii, oo], [filter_ldd, first])$, trans$(ii, oo, narrow_leq)\}$

When sending the problem [leq(c,d), in(c,[1,1]), in(d,[3,6]), leq(a,b), in(a,[1,10]), in(b,[3,6])] on port i, $filter\_ldd$ finds 2 candidates. The selector sends the candidate [leq(c,d), in(c,[1,1]), in(d,[3,6])] to the solver. This candidate cannot be modified by the solver. Thus, the selector sends the next candidate [leq(a,b), in(a,[1,10]), in(b,[3,6])] which is effectively reduced by the $narrow\_leq$ solver. The answer on o is finally [leq(a,b), in(a,[1,6]), in(b,[3,6], leq(c,d), in(c,[1,1]), in(d,[3,6])]. As in the bestApply primitive, we can change the strategy for applying cooperation by changing the Filter and Selector given as parameters.

Implementation of other primitives of the cooperation language of [6] with our components can be found in [12].

Fixed-point of a best_apply. We now use the bestApply pattern connected to a fixed-point pattern. In terms of cooperation, we compute the full reduction of a problem (the problem is locally consistent w.r.t. each constraint), and this can be done by computing the fixed-point of a bestApply.

When flattened, this pattern is composed of 45 components. When we send the problem [leq(a,b), leq(b,c), in(a,[7,8]), in(b,[1,9]), in(c,[2,7])] the system reduces domains of a, b, and c. We obtain the answer [leq(a,b), in(a,[7,7]), in(b,[7,7]), leq(b,c), in(c,[7,7])]. The trace of execution shows that the bestApply pattern was called 4 times: 3 times successfully (i.e., the solver really modified the problem), and once unsuccessfully (no modification of the problem); this determines that the fixed point is reached. The $narrow\_leq$ solver was called 7 times, and among them, 3 times successfully (this corresponds to the 3 successes of the bestApply). The other 4 times are tries that do not reduce the domains, i.e., either a candidate that fails in the bestApply, or the reach of the fixed-point.

6 Conclusion and Discussion

We have proposed a rule language for designing and implementing interactive components and basic coordination languages. In our language, concurrent rules manage interactions and applications of callback functions to a store of data that can freely be structured as an array, a list, a set of communication channels, etc.

In our rule language, we consider an implicit and freely structured store. Thus our rules do not need to show the complete store of data we are working on, whereas terms (stores in our language) must always be present in ELAN rules. In CHR, the store is implicit but fixed: it is a “set” of constraints whereas we can consider more diverse and complex structures.

Comparing our framework with a concurrent constraint programming (CCP, see e.g. [16]) language is not an easy task. First we can write different component languages, and second, there are numerous CC languages that have different properties or features (monotonicity of the store, one or several shared stores, different forms of ask and tell, ...). The component language presented in Section 5 could be seen as a CCCP (cooperative concurrent constraint programming language [6]) such that: there are several shared stores; stores are not monotonic (constraints can be added, removed, and modified), no solver is attached to stores (thus “ask” and “tell” are just “read” and “write”; the entailment is restricted to equality of ground terms); agents (our components) read at a given address in a/several stores (they read/write in a part of a/several stores, and this is closer to tuple based languages); agents (our components) can be connected to several stores, i.e., they can read and write in several stores (to our knowledge, no CC language has this feature).

In the future, we plan to investigate the possible connection of our language with the pi-calculus. Some preliminary work is in progress and we believe that components and patterns can be automatically expressed in the pi-calculus. The interest is that a solver cooperation written with components and patterns could be translated into pi-calculus expressions: then, some properties of solver cooperations could be verified, and cooperations could also be simplified.

We also plan to get an implementation of our rule language, and not only an implementation of a component language. Then, it will be easier to create and test new component languages.

Another point of interest is the dynamic creation of components. Up to now, a program is static; this means for example that a component cannot create a new component; or that a solver cooperation cannot launch a new solver or cannot adapt dynamically to the problem (e.g., by creating n cooperations to treat in parallel the n disjuncts of a problem). In terms of rules, this dynamic behavior implies a reflexive rule language. This also opens features such as mobility (in terms of CC): a component can change the store and the address where it is currently reading and writing.

References

1. K. R. Apt. The Essence of Constraint Propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999. Available via `http://arXiv.org/archive/cs/`.
2. F. Arbab and E. Monfroy. Coordination of Heterogeneous Distributed Cooperative Constraint Solving. *ACM SIGAPP Applied Computing Review*, 6(2):4–17, 1998.
3. F. Arbab. *Manifold2.0 reference manual.* CWI, Amsterdam, The Netherlands, May 1997.

[6] The authors thank P. Codognet for the idea of the acronym.

4. H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In *Logic programming: formal methods and practical applications*, Studies in Computer Science and Artificial Intelligence. Elsevier Science Publisher B.V., 1995.
5. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proc. of the 2nd Int. Workshop on Rewriting Logic and its Applications*, Pont–Mousson, France, 1998. Elsevier. E.N.T.C.S., vol. 15.
6. C. Castro and E. Monfroy. A Control Language for Designing Constraint Solvers. In *Proc. of the 3rd Int. Conference Perspective of System Informatics, PSI'99*, volume 1755 of *LNCS*, pages 402–415, Novosibirsk, Akademgorodok, Russia, 2000.
7. C. Castro and E. Monfroy. Basic Operators for Solving Constraints via Collaboration of Solvers. In *Proc. of the 5th Int. Conf. on A.I. and Symbolic Computation (AISC'2000)*, volume 1930 of *LNCS*, pages 142–146, Madrid, Spain, 2001. Springer.
8. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
9. L. Granvilliers, E. Monfroy, and F. Benhamou. Symbolic-interval cooperation in constraint programming. In *Proceedings of the 26th International Symposium on Symbolic and Algebraic Computation (ISSAC'2001)*, pages 150–166, University of Western Ontario, London, Ontario, Canada, 2001. ACM Press. ISBN 1-58113-417-7.
10. A. Kleymenov, D. Petunin, A. Semenov, and I. Vazhev. A model of cooperative solvers for computational problems. *Joint Bulletin of NCC and IIS. Series: Computer Science*, 16:115–128, 2001.
11. A. K. Mackworth. Constraint Satisfaction. In S.C. Shapiro, editor, *Encyclopedia of A.I.*, volume 1. Addison-Wesley Publishing Company, 1992. 2nd Edition.
12. E. Monfroy and C. Castro. Basic Components for Constraint Solver Cooperations. In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC'2003)*, Melbourne, Florida, USA, 2003. ACM Press.
13. E. Monfroy. The Constraint Solver Collaboration Language of BALI. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 211–230. Research Studies Press/Wiley, 2000.
14. G.A. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers – The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.
15. C. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In F. Baader and K. Schulz, editors, *Frontiers of Combining Systems (FroCoS'96)*, Applied Logic, pages 121–140. Kluwer Academic Publishers, 1996.
16. V. Saraswat. *Concurrent Constraint Programming.* MIT Press, Cambridge, London, 1993.

A Generic Trace Schema for the Portability of CP(FD) Debugging Tools[⋆]

Ludovic Langevine[1], Pierre Deransart[1], and Mireille Ducassé[2]

[1] Inria Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
{Ludovic.Langevine,Pierre.Deransart}@inria.fr
[2] Irisa/Insa, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
Mireille.Ducasse@irisa.fr

Abstract. Debugging tools are essential to help tune constraint solving programs and each platform has its environment tools. However, at present, these tools are specific and have to be redesigned and re-implemented for each constraint solver whereas much could be factorized. This article sets the foundations to enable debugging tools to be defined almost independently from finite domain solvers, and conversely, tracers to be built independently from these tools. We propose a *generic trace schema* based on a *generic observational semantics* which formalizes relevant aspects of constraint programming and solving. We illustrate the genericity of the schema on three representative families of finite domain solvers: CLP (Gnu-Prolog), CSP (Choco) and explanation based CP (PaLM). Debugging tools can use the generic trace and do not have to take into account all the details of the solvers. We experimented several combinations of new debugging tools with the above mentioned solvers. We show that they all find in the generic trace the whole information they require.

1 Introduction

There exists a number of useful debugging tools for finite domain solvers, for example, Grace [25], the Oz Explorer [27], the Oz Constraint Investigator [26], the CHIP global constraint visualizer [28], the Christmas Tree visualizer [4] or Clpgui [14]. However, not all solvers benefit from all the existing tools. Indeed, as illustrated by Fig. 1, at present if a debugging tool is to be integrated into a solver environment, the entire connection between the solver and the debugging tool has to be built, resulting in a set of one-to-one specialized connections between a solver and its tools. In general, hooks have to be added into the solver code in order to be able to send execution information to the debugging tools. Furthermore, the types of basic information required by a given debugging tool is not often made explicit and may have to be reverse-engineered. This is a non neglectable part of the cost of porting debugging tool from one constraint

[⋆] This work is partly supported by OADymPPaC, a French RNTL project: http://contraintes.inria.fr/OADymPPaC.

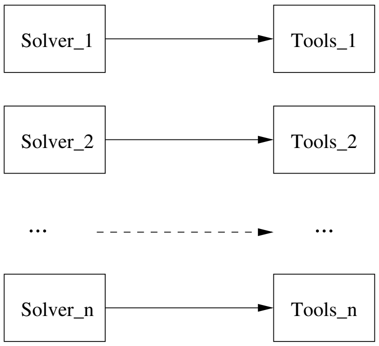


Fig. 1. Current situation: each solver has to be connected the hard way to a set of debugging tools

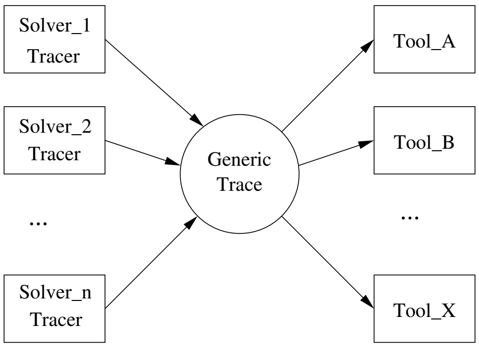


Fig. 2. Ideal situation: a generic trace schema enables all debugging tools to work straightforwardly with all solver tracers

programming platform to another. The connection between a solver and a debugging tool in itself is rather tedious to achieve and it has to be done for each solver and each debugging tool.

In order to solve the above mentioned problems, we advocate to make the debugging information explicit in terms of an execution trace schema adapted to finite domain solvers. Traces conforming to this schema can be generated by solver tracers and used by debugging tools[1]. As a matter of fact, even if the finite domain solvers have different implementations and different optimizations, their operational semantics is not so different. Hence, in an ideal world, as illustrated by Fig. 2, we would aim at specifying a generic trace schema so that all solver tracers generate traces following this schema and the generated traces would be straightforwardly understood by all debugging tools.

However, experience shows that the situation is slightly more complicated. As illustrated by Fig. 3, all tracers cannot generate exactly the same trace.

[1] Note that there exists mechanisms to avoid generating irrelevant pieces of trace [11].

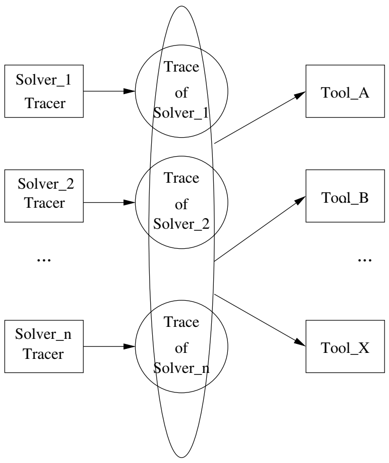


Fig. 3. Our approach: all debugging tools work for all solvers, using the generic part of the solver traces

Therefore the debugging tools, in order to work for different solvers, have to have a specialized front end. We show in the following that this front end can be considerably small for three independently developed debugging tools. The presented trace schema is indeed rich enough to cover the needs of at least all these tools. It should be noted that the tedious work of adding hooks into the solvers is done only once for each solver, while building its tracer. Connecting a debugging tool does not cost anything on the solver side.

In this article we specify a *generic trace schema.* The traces generated by a given tracer will follow a specialized version of this generic trace schema. In a first stage, we formally specify a *generic observational semantics* by a set of state transition rules. Each rule identifies a type of execution events which can be useful for debugging. The rules also define what part of the execution state is of interest for each event type. An observational semantics can be viewed as a sampling of a complete formal semantics.

Then as illustrated in Fig. 4 this generic observational semantics (GOS) is specialized for each solver S_k into a specialized observational semantics (OS_k). From the observational semantics the trace schemata are designed by abstracting the state information. Traditionally, a tracer does not explicitly give information about states when this information can be deduced. Note that, therefore, the rules are very important to interpret a trace, it is unfortunate that they usually do not explicitly appear in the documentation of tracers.

In the remaining of this article, the “arrows” of Fig. 4 are informally discussed and the “boxes” are formally described: from the generic observational semantics to three specialized observational semantics for GNU-Prolog, Choco and PaLM, and from the generic trace schema to the three specialized trace schemata for GNU-Prolog, Choco and PaLM. Three solvers and three debugging tools have

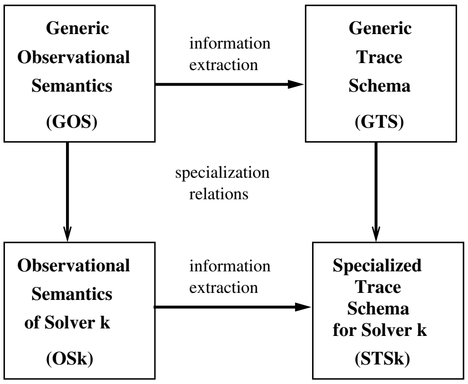

Generic Observational Semantics (GOS)
information extraction
Generic Trace Schema (GTS)
specialization relations
Observational Semantics of Solver k (OSk)
information extraction
Specialized Trace Schema for Solver k (STSk)

Fig. 4. From Observational Semantics to Trace Schemata (Top: *generic*, Bottom: *specialized*)

been connected together by different people. The generic trace schema has proven very effective for these experiment and has indeed enabled different partners of a national project to share data and tools. The front ends for the experiments have been easily implemented by the developers of each tool.

The contribution of this paper is therefore to propose a generic trace schema which required only minor specialization in order to be adapted to three quite different finite domain constraint solvers.

In the following, Section 2 gives the formal specification of the generic observational semantics. Section 3 specifies the generic trace schema. Section 4 gives the specializations of the semantics and of the schema for three different kinds of solvers. Section 5 describes the experimentation with three solvers and a meta-interpreter, and three debugging tools.

2 A Generic Observational Semantics for CLP(FD)

A trace is a sequence of execution events. Each event is an instance of a finite set of event types. An event-type corresponds to a transition from an execution state to another. The execution state is formalized as an *observed state* and each event type is defined by a state-transition rule between two observed states.

An observed state consists of two parts: the solver state and the search-tree state. In the remaining of this section, we describe in detail the components of the solver state, we present the search-tree and then give all the rules that formally describe the events. Events are divided into two classes: control and propagation. Control events are related to the management of variables and constraints, as well as the management of the search. Propagation events are related to the reduction of domains and the awakening process of constraints.

2.1 Solver State and Search-Tree

Definition 1 (Solver State). *A solver state is a 8-tuple:* $\mathbb{S} = (\mathcal{V}, \mathcal{C}, \mathcal{D}, A, E, R, S_c, S_e)$ *where:* $\mathcal{V}$ *is the set of already declared variables;* $\mathcal{C}$ *is the set of* *already declared constraints;* $\mathcal{D}$ *is the function that assigns to each variable in* $\mathcal{V}$ *its domain (a set of values in the finite set* $\mathbb{D}$*);* A *is the set of active couples of the form (constraint, solver event*[2]*);* E *is the set of entailed constraints;* R *is the set of unsatisfiable (or rejected) constraints.* S_c *is the set of sleeping constraints;* S_e *is the set of solver events to propagate ("sleeping events").*

A, S_c, E and R are used to describe four specific states of a constraint during the propagation stage: active, sleeping, entailed or rejected. These states are consistent with the states Müller defines to describe the *propagators* in the Oz system [26]. The *store of constraints* is actually the set of all the constraints taken into account. The store is called σ in the following and defined as the partition $\sigma = \{c \mid \exists(c,e) \in A\} \cup S_c \cup E \cup R$. All the constraints in σ are defined, thus $\sigma \subseteq \mathcal{C}$. The set of variables involved in the c constraint is denoted by $\mathbf{var}(c)$. The predicate $false(c, \mathcal{D})$ (resp. $entailed(c, \mathcal{D})$) holds when the constraint c is considered as unsatisfiable (resp. entailed) by the domains in $\mathcal{D}$.

The search is often described as the construction of a search-tree.

Definition 2 (Search-Tree State). *The search-tree is formalized by a set of labeled nodes* $\mathcal{N}$ *and a function* Σ *which assigns to each node a solver state. The nodes in* $\mathcal{N}$ *are ordered by the construction. Three kinds of nodes are defined and characterized by three predicates: failure leave (*$failed(\mathbb{S})$*), solution leave (*$solution(\mathbb{S})$*), and choice-point node (*$choice\text{-}point(\mathbb{S})$*). The last visited node is called* current node *and is denoted* ν*. The usual notion of* depth *is associated to the search-tree: the depth is increased by one between a node and its children. The function* δ *assigns to a node* n *its depth* $\delta(n)$*. Therefore, the state of the search-tree is a quadruple:* $\mathbb{T} = (\mathcal{N}, \Sigma, \delta, \nu)$*.*

In the initial solver state, ν denotes the root of the search-tree and all the sets that are part of $\mathbb{S}$ and $\mathbb{T}$ are empty.

2.2 Transition between Observed States

The observational semantics is specified by rules which characterize possible state transitions. The top of each rule describes the side conditions on the observed state $(\mathbb{S}, \mathbb{T})$ required to use the rule. The bottom of the rule details the change of the observed state. Some definitions are added on the right-hand side.

$$\textsf{name}\ \frac{conditions\_on\_current\_state(\mathbb{S}, \mathbb{T})}{actions\_on\_the\_observed\_state(\mathbb{S}, \mathbb{T})}\ \{definitions\}$$

In the following Sec. 2.3 describes the control events and Sec. 2.4 the propagation events.

[2] This work inherits from two areas, constraint solving and debugging, which both use the word "event" in correlated but different meanings: a *solver event* is produced by the solver and has to be propagated (e.g. the update of the domain bounds of a variable); a *trace event* corresponds to an execution step which is worth reporting about.

$$\text{new variable}\ \frac{v \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{v\},\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(v, D_v)\}}\ \{D_v : \text{initial domain of } v\}$$

$$\text{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \mathbf{var}(c) \subseteq \mathcal{V}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

$$\text{post}\ \frac{c \in \mathcal{C}\ \wedge\ c \notin \sigma}{A \leftarrow A \cup \{(c, \bot)\}}$$

$$\text{choice point}\ \frac{\mathit{choice\text{-}point}(\mathbb{S})\ \wedge\ n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{back to}\ \frac{n \in \mathcal{N} \wedge \mathit{choice\text{-}point}(\Sigma(n))}{\mathbb{S} \leftarrow \Sigma(n),\quad \nu \leftarrow n}$$

$$\text{solution}\ \frac{\mathit{solution}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{failure}\ \frac{\mathit{failure}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{remove}\ \frac{c \in \sigma}{\sigma \leftarrow \sigma - \{c\}}$$

$$\text{restore}\ \frac{v \in \mathcal{V}\quad \wedge\quad \Delta_v \cap \mathcal{D}(v) = \emptyset}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) \cup \Delta_v}\ \{\Delta_v \text{ is a subset of the initial } D_v\}$$

Fig. 5. Control rules of the generic observational semantics

2.3 Control

The rules of Fig. 5 describe the control part of the observational semantics. The control part handles the constraint store and drives the search. Rule new variable specifies that a new variable v is introduced in $\mathcal{V}$ and that its initial domain is D_v. Rule new constraint specifies that the solver introduces a new constraint c in $\mathcal{C}$, having checked that all variables involved in c are already defined. This constraint is declared without being posted: it does not yet belong to the store σ. The activation of a declared constraint c is specified by the rule post: the constraint is entered in the store as an active constraint and is ready to make domain reductions. It is attached to the event $\bot$ which denotes the activation of the constraint.

The following four rules describe the construction of the search-tree. Rule choice point specifies that the current solver state corresponds to a new node of the search-tree which is candidate as choice-point, i.e. the solver may jump back to this state later. This state is recorded in Σ. Jumping back from the current solver state to a previous choice-point is specified by the rule back to: a whole former state is restored in $\mathbb{S}$. Finally, two rules are used to declare the leaves of the search-tree. solution specifies that the current solver state corresponds to a solution leaf. failure specifies that the current solver state is a failure leaf.

Two additional rules are used to describe search strategies that are not based on a search-tree, such as the *path-repair* technique presented by Jussien and Lhomme [20]. Those strategies enable the removal of any constraint from the store σ and the cancellation of any previous domain reductions, leading to new

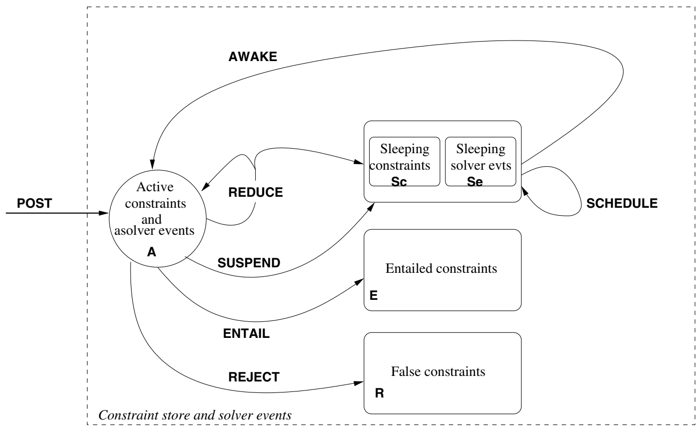


Fig. 6. Generic Observational Semantics: illustration of the transitions described by the propagation rules

parts of the search space. Rule remove removes a constraint c from the store σ. The rule restore specifies that the solver is restoring some values Δ_v in the domain of variable v.

2.4 Propagation

The propagation can be described by state transition rules acting in the solver state, as illustrated by Fig. 6. These rules are formalized by Fig. 7. A, S_c, S_e, E and R are parts of the current state $\mathbb{S}$ previously defined. The active pairs in A can reduce some domains because of their solver events. Rule reduce specifies that the solver reduces the domain of a variable of an active constraint attached to a solver event. A single domain can be reduced by this rule. $\bar{e}'$ is a set of solver events that characterize the reduction. Examples of solver events are "increase of the lower bound of the domain of x" or "instantiation of the variable y". Each domain reduction generates new solver events that are recorded in S_e. When an active pair cannot reduce any domain at the moment, the e event is said to have been propagated by c and the c constraint is suspended in S_c (rule suspend). An active constraint that is true is said to be *entailed* and put in E (rule entail). An active constraint that is unsatisfiable is said to be *false* and put in R by the reject rule. Solver events in S_e are waiting to be propagated through sleeping constraints in S_c. A solver dependent action schedules the propagation by acting on a sleeping constraint and a sleeping event ($action(c, e)$ in schedule): this leads to a modification of the internal structure of S_c and S_e. If the solver dependent condition $awakecond(c, e)$ holds, such a pair (c: *constraint to awake*, e: *awakening cause*) can then be activated by an awake transition. This may lead to new domain reductions. Notice that e can be $\perp$: the awakening is then due to the sole activation of the constraint.

$$\text{reduce}\ \frac{(c,e) \in A \ \wedge\ v \in \mathbf{var}(c)}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) - \Delta_v^c,\quad S_e \leftarrow S_e \cup \bar{e}'} \left\{ \begin{array}{l} \Delta_v^c \text{ is a subset of } \mathcal{D}(v) \text{ to remove} \\ \bar{e}' \text{ is a set of solver events on } v \end{array} \right\}$$

$$\text{suspend}\ \frac{(c,e) \in A}{A \leftarrow A - \{(c,e)\},\quad S_c \leftarrow S_c \cup \{c\}}$$

$$\text{entail}\ \frac{(c,e) \in A \ \wedge\ entailed(c,\mathcal{D})}{A \leftarrow A - \{(c,e)\},\quad E \leftarrow E \cup \{c\}}$$

$$\text{reject}\ \frac{(c,e) \in A \ \wedge\ false(c,\mathcal{D})}{A \leftarrow A - \{(c,e)\},\quad R \leftarrow R \cup \{c\}}$$

$$\text{awake}\ \frac{c \in S_c \ \wedge\ e \in S_e \cup \{\bot\} \ \wedge\ awakecond(c,e)}{A \leftarrow A \cup \{(c,e)\},\quad S_c \leftarrow S_c - \{c\}}$$

$$\text{schedule}\ \frac{c \in S_c \ \wedge\ e \in S_e \ \wedge\ action(c,e)}{S_c \leftarrow S_c',\quad S_e \leftarrow S_e'} \ \{S_c' \text{ and } S_e' \text{ are solver dependent}\}$$

Fig. 7. Propagation rules of the generic observational semantics

Table 1. Specific Attributes of the Ports

Control Ports		**Propagation Ports**	
new variable	$v,\ D_v$	reduce	$c,\ e,\ v,\ \bar{e}',\ \Delta_v^c$
new constraint	c	suspend, entail	c
post, remove	c	reject	c
restore	$v,\ \Delta_v$	awake	$c,\ e$
choice point	ν after the state transition	schedule	$c,\ e$
back to	ν before and after the state transition		
solution, failure	ν after the state transition		

3 Generic Trace Schema

The two sets of rules presented by Fig. 5 and Fig. 7 specify an observational semantics. A generic trace schema derives from this semantics. As already mentioned, a trace is a sequence of events. Each trace event corresponds to the application of a semantic rule. Therefore each semantic rule gives its name to a type of trace event: a *port*.

Definition 3 (Generic Trace). *A generic trace is a sequence of trace events consisting of a set of attributes: a sequential event number; the depth in the search tree; the port (the type of the corresponding semantic rule); the whole observed state of the solver after the transition; some specific attributes depending on the port.*

The specific attributes correspond to conditions and characteristics of specific actions performed by each rule. For example, the port reduce has additional attributes corresponding to the concerned constraint and solver event, the variable v that is being reduced, the value withdrawal Δ_v^c and the solver events that these reductions generate, $\bar{e}'$. Table 1 presents the list of the specific attributes for each rule, using the notations of Fig. 5 and Fig. 7.

4 Specialized Observational Semantics

This section specializes the generic observational semantics for three solvers: the constraint logic programming platform GNU-Prolog, the constraint programming library Choco and the explanation-based platform PaLM. It shows how the generic observational semantics fits those different platforms in such a way that each specialization satisfies the semantics of a platform. In this section, the specialized observed state of each platform is detailed, as well as its dedicated control and propagation rules.

4.1 Specialized Observational Semantics for GNU-Prolog

GNU-Prolog is a Prolog platform with an embedded FD solver developed by Diaz [10]. A self-contained description of the specialized observational semantics, as well as the description of the actual tracer implementation, can be found in [22].

Solver State for GNU*-Prolog.* A GNU-Prolog primitive constraint is defined by several indexicals (constraints of the form $x \in f(\mathcal{D}_{|\mathbf{var}(c)})$). The constraints considered in GNU-Prolog are the primitive constraints whereas the indexicals are *propagators*. The solver state of GNU-Prolog is formalized as $\mathbb{S} = (\mathcal{V}, \mathcal{C}, \mathcal{D}, A_c, A_e, E, R, S_c, Q)$. A_c contains at most a single active constraint while A_e contains at most the current event to propagate. The generic set A is thus defined as $A = A_c \times A_e$. The set S_e of the generic model is refined as $S_e = Q \cup A_e$ where Q is the tail of the propagation queue that drives the constraint awakenings, and A_e is its head.

Control of GNU*-Prolog Executions.* As defined in Standard Prolog [8], the search strategy of GNU-Prolog is modeled as the depth-first left-to-right traversal of a tree. From a CP point of view, the search-tree consists of a subset of nodes of the SLD tree, the traversal order being preserved. Therefore the search is described by the four events choice point, back to, solution and failure only. Since the search is completely described as a tree, the remove and restore rules are useless. Notice that, in GNU-Prolog, failure denotes FD constraint rejections as well as classical Prolog failures. new variable denotes the creation of a FD variable (a logic variable that appears in a constraint and gets a domain). new constraint followed by post denotes the call of a GNU-Prolog goal that is a primitive constraint. The only different rules with respect to the control part of the generic semantics are post (because of $A = A_c \times A_e$) and solution (there can be no solution once R is not empty).

Propagation. The propagation in GNU-Prolog is formalized by the rules of Fig. 10 and illustrated in Fig. 8. The constraint propagation is driven by the Q queue: each time a domain is reduced, the corresponding domain updates are recorded

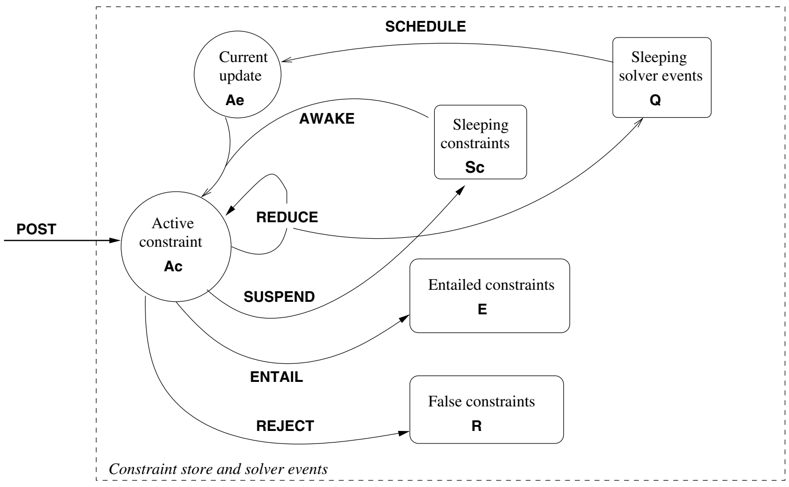


Fig. 8. GNU-Prolog Specialized Observational Semantics: illustration of the transitions described by the propagation rules

in Q (rule reduce). Updates in Q are dequeued one by one and put into A_e (rule schedule). The solver event in A_e awakes each constraint that it potentially impacts (rule awake). In GNU-Prolog, the potential impact of e on c is modeled by dependency lists denoted by the relation $dependence(c, e)$.

A solver event e is represented by a couple (v, t) where v is the reduced variable and t is the type of reduction. The types of reduction are: min (increase of the lower bound), max (decrease of the upper bound), $minmax$ (either of the two previous ones), val (instantiation of the variable) and dom (any domain reduction). A single reduce can then lead to several solver events. For example, when $\mathcal{D}(v)$ is reduced from $\{1, 2, 3\}$ to $\{1, 2\}$ the resulting solver events are $\bar{e} = \{(v, min), (v, minmax), (v, dom)\}$.

GNU-Prolog tests constraint entailment only when a constraint is posted $(A_e = \{\bot\})$ (entail).

A constraint is said to be rejected when the computed inconsistent values of a variable v (Δ_v^c) is the whole domain of v (reject). Notice that, once a constraint is rejected, no propagation rule can be applied anymore.

4.2 Specialized Observational Semantics for Choco

Choco is an object-oriented library of constraint programming designed by Laburthe et al. [21].

Solver State for Choco. The state of the Choco solver can be mapped onto the generic model with the refinement $S_e = Q_h \cup Q_t$ and $E = \emptyset$ (Choco does not detect constraint entailment). Thus, $\mathbb{S} = (\mathcal{V}, \mathcal{C}, \mathcal{D}, A, R, S_c, Q_h, Q_t)$. During the propagation, A contains at most a pair. This active pair consists of the

$$\textsf{new variable}\ \frac{v \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{v\},\ \ \mathcal{D} \leftarrow \mathcal{D} \cup \{(v, D_v)\}}\ \{D_v : \text{initial domain of } v\}$$

$$\textsf{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \mathbf{var}(c) \subseteq \mathcal{V}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

$$\textsf{\textit{post}}\ \frac{c \in \mathcal{C}\ \wedge\ c \notin \sigma}{A_c \leftarrow \{c\},\ \ A_e \leftarrow \{\bot\}}$$

$$\textsf{choice point}\ \frac{\mathit{choice\text{-}point}(\mathbb{S})\ \wedge\ n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\ \ \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\ \ \nu \leftarrow n}$$

$$\textsf{back to}\ \frac{n \in \mathcal{N} \wedge \mathit{choice\text{-}point}(\Sigma(n))}{\mathbb{S} \leftarrow \Sigma(n),\ \ \nu \leftarrow n}$$

$$\textsf{\textit{solution}}\ \frac{\mathit{solution}(\mathbb{S}) \wedge n \notin \mathcal{N} \wedge R = \emptyset}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\ \ \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\ \ \nu \leftarrow n}$$

$$\textsf{failure}\ \frac{\mathit{failure}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\ \ \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\ \ \nu \leftarrow n}$$

Fig. 9. Specialized control rules for GNU-Prolog

constraint of which one filtering algorithm is running and the domain update that is propagated, or $\bot$. As in GNU-Prolog, S_e (the set of all solver events to process) is a queue that is refined as the head Q_h (event currently processed) and the tail Q_t. These events are processed in turn in order to schedule the constraint awakenings.

Control. The control part of the specialized semantics is shown in Fig. 12. Choco search strategy is modeled by a tree where any previous choice point can be restored at backtracking. This is taken into account by the generic back to rule as soon as each choice point has been declared by the generic choice point rule. The failure leaves are always due to a constraint rejection, thus $R \neq \emptyset$. Since the search can be described as a tree, the remove and restore rules are useless.

Propagation. Choco's propagation engine is illustrated in Fig. 11 and formalized in Fig. 13. The filtering algorithms of a given constraint are defined by several methods (in the sense of object programming). For instance, `awakeOnInf` propagates a lower bound increase and `awakeOnInst` propagates the instantiation of a variable. The awakening of a constraint is the invocation of one of these methods. The propagation is driven by the queue of the solver events to propagate. schedule focuses on an event e in the queue Q_t and puts it in Q_h as formalized in Fig. 13. The event in Q_h is processed by awakening every constraint that can propagate it, according to the specialized relation $\mathit{dependence}(c, e)$ (awake). Notice that a constraint can be awaken with the $\bot$ event: it denotes the invocation of a specific filtering algorithm that does not take into account any domain update but only the current domains. The value removals are computed knowing e (Δ_v^c becomes $\Delta_v^{c_e}$ in the rule reduce to express that). A constraint is rejected only when it has just emptied out a domain $\mathcal{D}(v) = \emptyset$ (reject). Once the processing of an event e is complete, another event can be selected, until Q_t is empty.

$$\mathit{reduce}\ \frac{A_c = \{c\}\ \wedge\ v \in \mathbf{var}(c)\ \wedge\ R = \emptyset}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) - \Delta^c_v,\quad Q \leftarrow Q \cup \bar{e}'}\ \left\{\begin{array}{l}\Delta^c_v \text{ is a subset of } \mathcal{D}(v) \text{ to remove}\\ \bar{e}' \text{ is a set of solver events on } v\end{array}\right\}$$

$$\mathit{suspend}\ \frac{A_c = \{c\}\ \wedge\ R = \emptyset}{A_c \leftarrow \emptyset,\quad S_c \leftarrow S_c \cup \{c\}}$$

$$\mathit{entail}\ \frac{A_c = \{c\}\ \wedge A_e = \{\bot\}\ \wedge\ \mathit{entailed}(c, \mathcal{D})\ \wedge\ R = \emptyset}{A_c \leftarrow \emptyset,\quad E \leftarrow E \cup \{c\}}$$

$$\mathit{reject}\ \frac{A_c = \{c\}\ \wedge\ v \in \mathbf{var}(c) \wedge \Delta^c_v = \mathcal{D}(v)}{A_c \leftarrow \emptyset,\quad R \leftarrow \{c\}}$$

$$\mathit{awake}\ \frac{A_c = \emptyset\ \wedge\ A_e = \{e\} \wedge c \in S_c\ \wedge\ \mathit{dependence}(c, e)\ \wedge\ R = \emptyset}{A_c \leftarrow \{c\},\quad S_c \leftarrow S_c - \{c\}}$$

$$\mathit{schedule}\ \frac{A_c = \emptyset \wedge\ e \in Q\ \wedge\ R = \emptyset}{A_e \leftarrow \{e\},\quad Q \leftarrow Q - \{e\}}$$

Fig. 10. Specialized propagation rules for GNU-Prolog

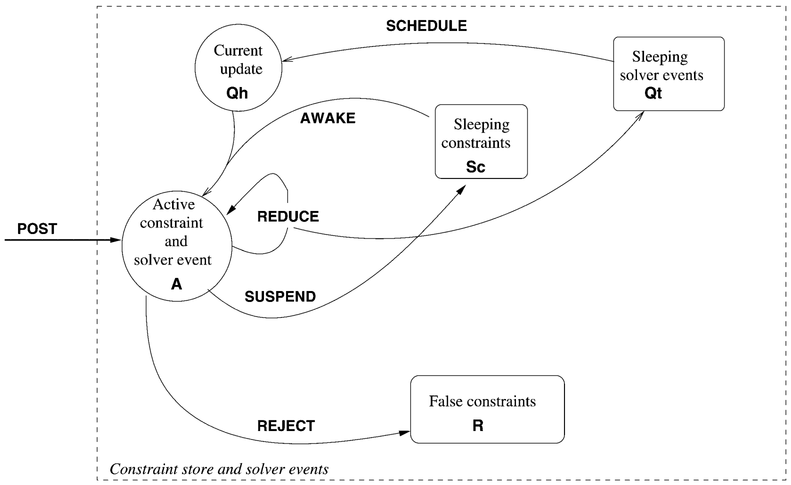

SCHEDULE
Current update Qh
Sleeping solver events Qt
AWAKE
Sleeping constraints Sc
POST
Active constraint and solver event A
REDUCE
SUSPEND
REJECT
False constraints R
Constraint store and solver events

Fig. 11. Choco Specialized Observational Semantics: illustration of the transitions described by the propagation rules

4.3 Specialized Observational Semantics for PaLM

PaLM is an explanation-based constraint programming platform by Jussien and Barichard [19]. It is a major extension of the Choco library presented above. Each value removal is labeled with a set of constraints that makes the value inconsistent. Such a set of constraints is called an *explanation* of the removal. The function that assigns to each value removal (v, d) (removal of d from $\mathcal{D}(v)$) its explanation is denoted $\mathcal{E}(v, d)$. The explanations are computed by the filtering algorithms, knowing the explanations of the previous value removals. This implementation of explanations is consistent with the theoretical framework developed by Debruyne et al. [6]. Explanations enable the solver to relax a constraint while

$$\textsf{new variable}\ \frac{v \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{v\},\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(v, D_v)\}}\ \{D_v : \text{initial domain of } v\}$$

$$\textsf{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \mathbf{var}(c) \subseteq \mathcal{V}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

$$\textsf{post}\ \frac{c \in \mathcal{C}\ \wedge\ c \notin \sigma}{A \leftarrow \{(c, \bot)\}}$$

$$\textsf{choice point}\ \frac{\mathit{choice\text{-}point}(\mathbb{S})\ \wedge\ n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \varSigma \leftarrow \varSigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\textsf{back to}\ \frac{n \in \mathcal{N} \wedge \mathit{choice\text{-}point}(\varSigma(n))}{\mathbb{S} \leftarrow \varSigma(n),\quad \nu \leftarrow n}$$

$$\textsf{\textit{solution}}\ \frac{R = \emptyset \wedge \mathit{solution}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \varSigma \leftarrow \varSigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\textsf{\textit{failure}}\ \frac{R \neq \emptyset \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \varSigma \leftarrow \varSigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

Fig. 12. Specialized control rules for Choco.

$$\textsf{\textit{reduce}}\ \frac{A = \{(c, e)\} \wedge v \in \mathbf{var}(c) \wedge \varDelta_v^{ce} \neq \emptyset \wedge R = \emptyset}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) - \varDelta_v^{ce},\quad Q_t \leftarrow Q_t \cup \{e'\}}\ \left\{\begin{array}{l}\varDelta_v^{ce}\text{: values that } c \text{ has} \\ \text{to remove from } \mathcal{D}(v) \\ \text{because of } e\text{. The re-} \\ \text{duction generates } e'.\end{array}\right\}$$

$$\textsf{\textit{suspend}}\ \frac{A = \{(c, e)\}\ \wedge\ R = \emptyset}{A \leftarrow \emptyset,\quad S_c \leftarrow S_c \cup \{c\}}$$

$$\textsf{\textit{reject}}\ \frac{A = \{(c, e)\}\ \wedge\ v \in \mathbf{var}(c) \wedge \mathcal{D}(v) = \emptyset}{A \leftarrow \emptyset,\quad R \leftarrow \{c\}}$$

$$\textsf{\textit{awake}}\ \frac{A = \emptyset \wedge c \in S_c \wedge e \in Q_h \cup \{\bot\} \wedge \mathit{dependence}(c, e) \wedge R = \emptyset}{S_c \leftarrow S_c - \{c\},\quad A \leftarrow \{(c, e)\}}$$

$$\textsf{\textit{schedule}}\ \frac{A = \emptyset \wedge e \in Q_t \wedge R = \emptyset}{Q_h \leftarrow \{e\},\quad Q_t \leftarrow Q_t - \{e\}}$$

Fig. 13. Specialized propagation rules for Choco.

undoing the value removals it is responsible for. When a domain has just been wiped out, the resulting failure can be diagnosed thanks to the recorded explanations. PaLM is then able to move to another point of the search space by relaxing well-chosen constraints.

PaLM specialization of the propagation part is presented before the control part, because the first one introduces several details about explanations that are worth knowing to understand the control well.

Solver State for PaLM. The solver state of PaLM is modeled as the state of Choco, with the additional explanation function $\mathcal{E}$. Thus $\mathbb{S} =< \mathcal{V}, \mathcal{D}, \mathcal{C}, A, R, S_c, Q_h, Q_t, \mathcal{E} >$, we have again $S_e = Q_h \cup Q_t$ with Q_h a singleton. In PaLM there is no detection of constraint entailment, therefore the set of the entailed constraints E is not used and the entail rule is irrelevant. $\mathcal{E}$

$$\textsf{reduce}\ \frac{A=\{(c,e)\}\ \wedge\ v\in\mathbf{var}(c)\ \wedge\ \Delta_v^{ce}\neq\emptyset\ \wedge\ C\subseteq\sigma\ \wedge\ R=\emptyset}{\mathcal{D}(v)\leftarrow\mathcal{D}(v)-\Delta_v^{ce},\quad Q_t\leftarrow Q_t\cup\{e'\},\quad \mathcal{E}\leftarrow\mathcal{E}\cup\{(v,d,C)\mid d\in\Delta_v^{ce}\}}$$

$$\left\{\begin{array}{l}\Delta_v^{ce}\text{ set of inconsistent values for }v\\ C\text{ explains the removal of }\Delta_v^{ce}\text{ from }\mathcal{D}(v)\\ \text{The reduction generates }e'\end{array}\right\}$$

$$\textsf{suspend}\ \frac{A=\{(c,e)\}\ \wedge\ R=\emptyset}{A\leftarrow\emptyset,\quad S_c\leftarrow S_c\cup\{c\}}$$

$$\textsf{reject}\ \frac{A=\{(c,e)\}\ \wedge\ v\in\mathbf{var}(c)\wedge\mathcal{D}(v)=\emptyset}{A\leftarrow\emptyset,\quad R\leftarrow\{c\}}$$

$$\textsf{awake}\ \frac{A=\emptyset\wedge c\in S_c\wedge e\in Q_h\cup\{\perp\}\wedge \mathit{dependence}(c,e)\wedge R=\emptyset}{S_c\leftarrow S_c-\{c\},\quad A\leftarrow\{(c,e)\}}$$

$$\textsf{schedule}\ \frac{A=\emptyset\wedge e\in Q_t\wedge R=\emptyset}{Q_h\leftarrow\{e\},\quad Q_t\leftarrow Q_t-\{e\}}$$

Fig. 14. Specialized propagation rules for PaLM

formalizes the recorded explanations. As presented above, it is a partial function $\mathcal{E}:\mathcal{V}\times\mathbb{D}\longrightarrow\mathcal{P}(\sigma)$ which assigns to each value removal (v,d) a set of non relaxed constraints that explains the removal. This partial function is updated only by the reduce and restore rules.

Domain reductions and propagation. The specialized propagation rules are presented in Fig. 14. Rules suspend, reject, awake and schedule are inherited from Choco. Explanations are introduced in the reduce rule. An elementary reduction is called *filtering decision* in PaLM terminology (for instance updating a bound or removing a specific value). As in Choco, it produces a single solver event e. This decision is taken by (one of) the filtering algorithm(s) of the active constraint in A and is explained by a single set of constraints C. C is computed by the filtering algorithm. While the inconsistent values are removed, the new solver event e is put in the propagation queue and the C explanation is recorded in $\mathcal{E}$ (for each removed value).

The reduction described by the reduce rule is made by c and is likely a consequence of some previous reductions of some variables involved in c. Therefore, C consists of c and the union of k other explanations ($k\geq 0$) about the variables involved in c (restriction denoted $\mathcal{E}_{|\mathbf{var}(c)}$). This additional description of C is formalized as follows:

$$\exists(C_1,\ldots,C_k)\subseteq\mathcal{E}^k_{|\mathbf{var}(c)}\text{ such that }C=\{c\}\cup\bigcup_{i=1}^{k}C_i$$

Control. The search in PaLM is not based on standard backtracking and there is no notion of search-tree. The rule back to is thus useless. The navigation in the search-space is fully described by the remove and restore rules specialized in Fig. 15. The choice point rule is used to tag specific points of the search-space where decisions have to be taken by the search-procedure. This is similar to

$$\textsf{new variable}\ \frac{v \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{v\},\ \ \mathcal{D} \leftarrow \mathcal{D} \cup \{(v, D_v)\}}\ \{D_v \text{ : initial domain of } v\}$$

$$\textsf{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \mathbf{var}(c) \subseteq \mathcal{V}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

$$\textsf{post}\ \frac{c \in \mathcal{C}\ \wedge\ c \notin \sigma}{A \leftarrow \{(c, \bot)\}}$$

$$\textsf{choice point}\ \frac{\mathit{choice\text{-}point}(\mathbb{S})\ \wedge\ n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\ \ \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\ \ \nu \leftarrow n}$$

$$\textsf{\textit{solution}}\ \frac{R = \emptyset \wedge \mathit{solution}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\ \ \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\ \ \nu \leftarrow n}$$

$$\textsf{\textit{failure}}\ \frac{R \neq \emptyset \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\ \ \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\ \ \nu \leftarrow n}$$

$$\textsf{remove}\ \frac{c \in \sigma}{\sigma \leftarrow \sigma - \{c\}}$$

$$\textsf{\textit{restore}}\ \frac{v \in \mathcal{V} \wedge R_v \subseteq \{d \in \mathbb{D} | \mathcal{E}(v, d) - \sigma \neq \emptyset\} \wedge E = \{\mathcal{E}(v, d) | d \in R_v\}}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) \cup R_v,\ \ Q_t \leftarrow Q_t \cup \bar{e},\ \ \mathcal{E} \leftarrow \mathcal{E} - E}$$
$$\{\bar{e} \text{ describes the restoration of } \mathcal{D}(v)\}$$

Fig. 15. Specialized control rules for PaLM

the notion of choice-point in a search-tree. As a result, $\mathcal{N}$ denotes the explored search-space, that is not necessarily a tree. Rules new variable, new constraint, post, choice point, solution and failure are inherited from Choco.

PaLM uses explanations when a failure occurs. The C recorded explanation of the empty domain is a *contradiction*. At least one constraint in C has to be relaxed. $\mathcal{E}$ enables PaLM to relax a constraint while undoing its previous effects: each value removal whose explanation contains the relaxed constraint have to be undone. The event consisting in the relaxation of a single constraint c is remove, as defined in the generic model. The restoration of the values R_v in $\mathcal{D}(v)$ is described by the rule restore. The removal of the R_v values from $\mathcal{D}(v)$ is explained by (at least) one removed constraint. As proven in [6], after domain restorations, a consistency-checking has to be done so as to get back to a locally coherent state as if the removed constraints never appeared in σ. This consistency-checking is done by re-propagating all the constraints involving the restored variable: each restoration $\bar{e}$ is then recorded into Q_t. A stage of domain restoration is thus followed by a sequence of propagation events until Q_h is empty or a new contradiction is found.

5 Experimental Validation

We started to validate the generic trace schema by connecting three independently developed debugging tools to four independently developed solvers with tracers.

The finite domain solvers which have been considered up to now are the three solvers discussed so far (GNU-Prolog, Choco and PaLM) and an instrumented meta-interpreter (Mescaline). Their tracers have been developed by two different

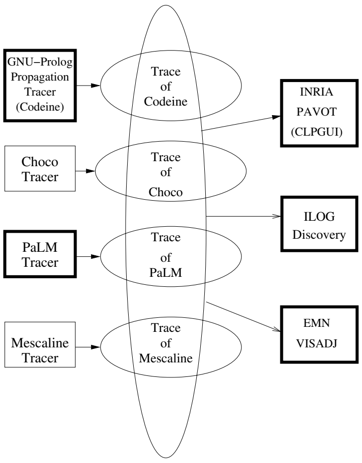


Fig. 16. Experimentation with 4 Solvers and 3 Tools (bold boxes are used in the illustration of Sec. 5.3)

teams: GNU-Prolog and Mescaline at INRIA, and the others at Ecole des Mines de Nantes (EMN).

The debugging tools offer several kinds of static or dynamic views of the traces. Three tools have been considered: PAVOT, a simple search-tree/propagation visualizer, VISADJ, a tool for adjacency matrix visualization, and Discovery a data analysis tool. All these tools allow large problems with hundreds of variables and constraints to be analyzed. All have been developed by different teams (resp. INRIA, EMN and ILOG).

All combinations are possible, as illustrated by Fig. 16. As explained below these tools have a general interface which can be parametrized by the user according to the origin of the trace.

The specialized traces produced by each tracer contain the ports (event types) and attributes defined in the generic trace schema[3]. In practice, tracers allow a trace to be stored in a file such that a post-mortem analysis of the trace is possible with different tools. The traces contain all the ports relevant to the corresponding solver but may sometime differ on the attributes. For example some events of the traces of PaLM contain an explanation attribute, when traces of other platform do not. A simple parametrization of the tools (selecting the events and attributes to be considered) allows specialized ports and attributes to be taken into account. In any case no modification of the generic trace scheme was necessary.

5.1 The Tracers

GNU-**Prolog (Codeine).** Codeine [24, 22] is an extension of GNU-Prolog [10] with the search-tree control and propagation tracer developed at INRIA. It is

[3] The trace is broadcasted according to an XML DTD defined in [15].

efficient and it has all the ports of the generic trace schema except remove and restore, according to the specialisation described in Sec. 4.1.

Choco [21], whose tracer has been implemented at EMN, contains similar finite domain solvers using usual backtracking which can be visualised by search-tree. Its trace has the same ports as above (cf. Sec. 4.2).

PaLM [19] is an explanation based solver developed at EMN. The control in PaLM is based on the manipulation of "explanations" instead of search-tree. Even if a search-tree can be theoretically derived from the solver behaviour, it is not explicitly described in the trace. Therefore there is no back to port. As PaLM does not check constraint entailment, there is no entail port either (cf. Sec. 4.3).

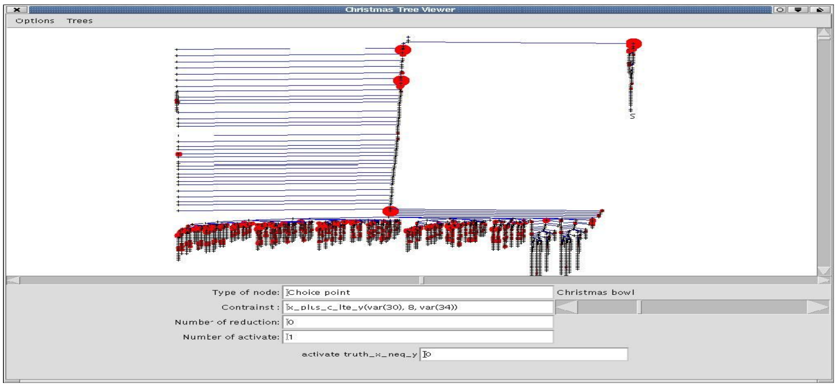
Mescaline [23] is a meta-interpreter embedded in Eclipse [1]. It is a generic tracer prototype written in Prolog and only small constraint programs can be executed. Implemented at INRIA in 2001 it has been used at the beginning of the OADymPPaC project to help specify the generic trace schema. It has all the ports of the generic trace schema except remove and restore.

5.2 The Debugging Tools

We have experimented with two categories of debugging tools: one is specifically designed to observe constraint solvers search-tree and propagation (PAVOT), two are general purpose data analysis tools whose interface has been adapted to handle a generic trace as a particular data entry (Discovery, VISADJ). Our purpose here is not to describe the tools, their efficiency or their capabilities, but to focus on their connectivity. The figures are intended to illustrate one of the functionalities of the tools.

PAVOT [16] is a propagation viewer currently developed at INRIA following the ideas of the Christmas-tree view of [4] and of CLPGUI [14] (in particular its 3-D viewer). A view represents a search-tree whose nodes are weighted by the number of selected events induced by the constraints posted just before the choice node. The weight is displayed in the form of circles whose radius is proportional to the weight. PAVOT uses the architecture described below for VISADJ, and supports all kinds of traces. However, to be used on PaLM traces a small adaptation was necessary: as PaLM does not have a search-tree, new nodes are created each time a constraint is declared (port new constraint). The resulting tree is simply linear. PAVOT allows post-mortem trace analysis as well as on the fly analysis, using a communication by sockets.

Fig. 17 shows the complete search-tree corresponding to the resolution of the bridge scheduling problem, minimizing the time to build a 5-segment bridge [29]. The picture shows the first solution reached quickly (top left part of the tree) with few propagations, then a large tree corresponding to the proof of optimality.



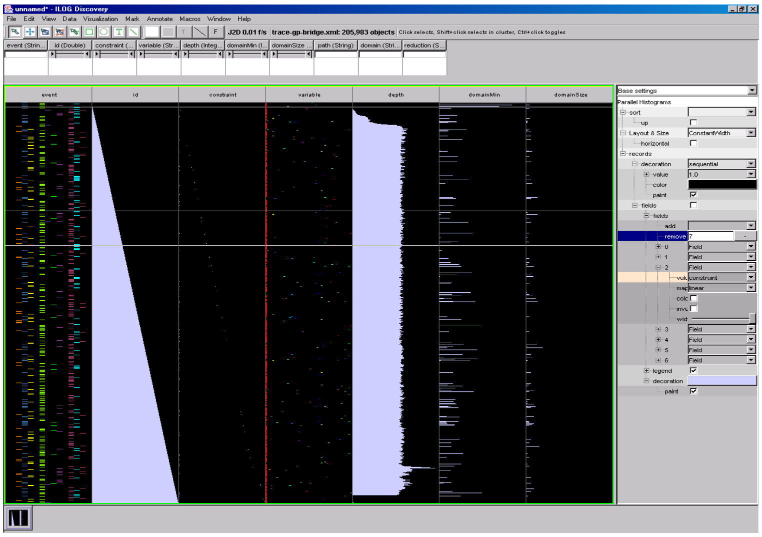
Type of node: Choice point
Contrainst : x_plus_c_lte_y(var(30), 8, var(34))
Numbe- of reduction: 0
Number of activate: 1
activate truth_x_neq_y 0

Fig. 17. Bridge scheduling problem: a 2-D view of the search space with PAVOT

Discovery ^{TM}[3] is a general purpose data analysis tool based on data manipulation and visualization techniques developed by ILOG. It has a special entry to read a trace conforming to the generic trace schema [15], and facilities to display predefined views (starfield display, parallel histograms, table lenses, treemaps, and user defined displays). At the moment, only post-mortem trace analysis is possible. The user selects which ports and attributes must be selected from the trace: event number, port, constraints, variables, path of the search-tree, min/max domain value, domain size. Explanations are not handled yet. The tool has been developed with the sole knowledge of the generic trace schema and could be used on all traces including those generated by PaLM. By selecting the data to analyze, the users make themselves the adaptation of the tool to the considered solver.

Fig. 18 shows a Discovery parallel histogram window for the bridge scheduling problem. It has seven columns and as many lines as trace events. The events are sorted by their sequential event number (second column). Each column corresponds respectively to event port (one colour per event type), sequential event number, concerned constraint, concerned variable, depth, domain min, and domain size. By the evolution of the depth from top to bottom one recognizes the profile of the search-tree depicted in Fig. 17. Discovery allows display to be manipulated (for example ordering wrt each column) and thus to discover more characteristics of the computation.

VISADJ has been designed at EMN to analyze large graphs, with several graph visualization techniques [17], including animations. It consists of several interconnected viewers, one devoted to showing graphs using adjacency matrix technique. An n-nodes graph is represented by a n-n-matrix whose elements are weighted edges. The tool helps analyze the structure of a graph whose nodes are usually constraints or variables and whose edges are coloured according to their weight (dark colours correspond to higher weight). The colour of an edge is thus an indication of how much a constraint influences another one. Fig. 19

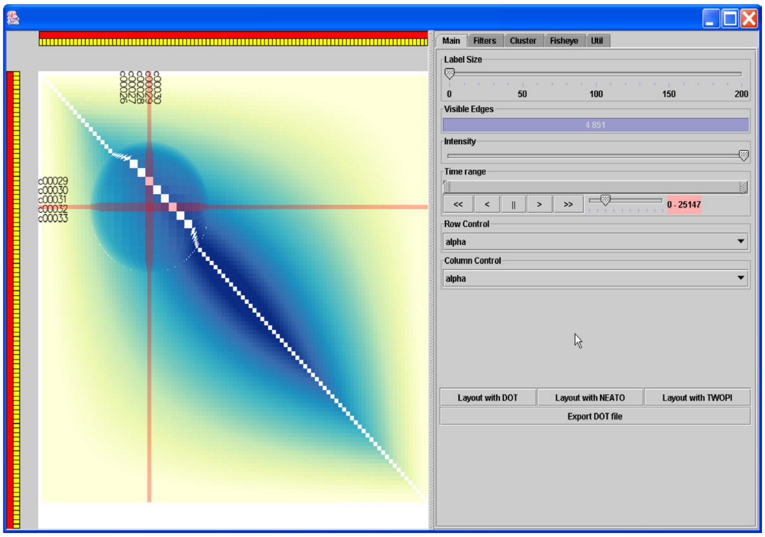

unnamed* - ILOG Discovery

J2D 0.01 f/s trace-gp-bridge.xml: 205,983 objects

Click selects, Shift+click selects in cluster, Ctrl+click toggles

File Edit View Data Visualization Mark Annotate Macros Window Help
event id constraint variable depth domainMin domainSize
Base settings
Parallel Histograms
sort
Layout & Size ConstantWidth
records
decoration sequential
fields
legend

Fig. 18. Bridge scheduling problem: a parallel histogram display by Discovery analysing the trace events



Label Size: 0 50 100 150 200

Visible Edges: 4 851

Intensity

Time range: 0 - 25147

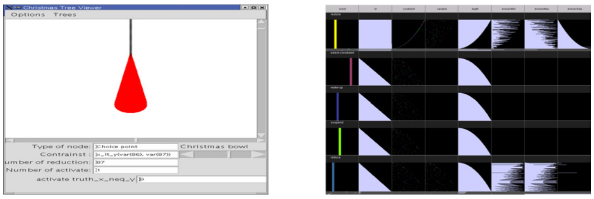
Row Control: alpha

Column Control: alpha

Fig. 19. An adjacency matrix for sorted(100) by the VISADJ tool. A fish-eye allows detailed analysis of a part of the graph

illustrates one of such large graphs whose detailed structure can by observed with the use of a “fish-eye”. This view represents the adjacency matrix built with the hundred posted constraints in the sorted(100) problem[4] solved with PaLM. The

[4] Hundred variables on domain [1..100] must different and increasingly ordered.



Type of node: Choice point
Contrainst : x_lt_y(var(86), var(87))
umber of reduction: 87
Number of activate: 1
activate truth_x_neq_y 0

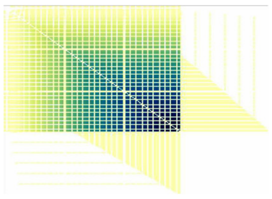


Fig. 20. A Gnu-Prolog trace of sorted(100) viewed with 3 tools: PAVOT, Discovery and VISADJ

constraints are ordered in the posting sequence from left to right and from top to bottom. A fish-eye allows the user to magnify a region of interest. Relationship between two constraints depends on the number of times both appear in an explanation. Notice that the farther the constraints are (in the posting sequence), the less they interact. Moreover, central constraints seem to interact with each other more often than peripheral ones (darker region in the centre of the view).

The board on the right allows in particular to create animations by defining an animated sequence of views corresponding to slices of the trace (in particular between two solutions). The tool has been originally experimented with traces issued from PaLM, and it has been interesting to see that it was able to handle traces generated by Codeine.

This Java based tool has essentially two components [17]: a library of "listeners" to access the trace and a library of viewers. The first component acts as a general interface able to interpret all events defined in the generic trace schema.

5.3 Using Several Tools on several Solvers

We finish this section with some views showing the versatility of our approach. The generic trace schema has been to be sufficiently complete to cover the debugging needs we have encountered so far. In fact all tools can easily be used with all generated traces of all the solvers provided the post mortem trace is sufficiently rich.

The following two series of views correspond to traces of *sorted(100)* respectively issued from Codeine (for Gnu-Prolog) and PaLM. Each view is intended to analyze the propagation with respectively PAVOT, Discovery and VISADJ. We just compare Gnu-Prolog (Fig. 20) and PaLM (Fig. 21) traces since these solvers are based on very different principles.

In Gnu-Prolog (Fig. 20-left) some propagation steps happen each time a new constraint is posted and the number of reduce events is proportional to the number of already posted constraints. This is clearly revealed by PAVOT where the diameter of the red balls increases regularly.

On Discovery (Fig. 20-middle) the parallel histogram display has been split by event types. Five ports are visible here, from top to bottom: back to, schedule, awake, suspend and reduce. The columns are the same as in Fig. 18. By the "depth" (column 5) we immediately observe the complementary roles played by the ports back to and reduce: a linear tree is developed, then the whole backtracking occurs

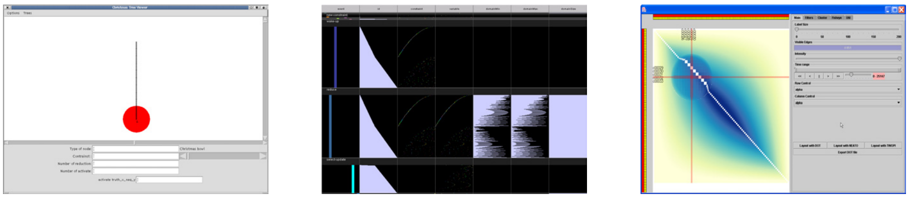


Fig. 21. A PaLM trace of sorted(100) viewed with 3 tools: PAVOT, Discovery and VISADJ

at the end (high event numbers in column 2 for back to). We observe the similarities between the events schedule, awake and suspend which are indistinguishable on this problem (same displays). Finally we observe the regularity of the variable domains reductions (column 7).

The presented VISADJ view (Fig. 20-right) is the adjacency matrix of the hundreds posted constraints. As there is no explanation in the traces of Codeine the level of interaction between two constraints corresponds to the number of events, generated by a constraint, which influence another one. We observe that a new posted constraint interacts with all previously posted ones.

Fig. 21 shows the same views as in Fig. 20 but obtained with traces generated by PaLM instead of Codeine. This allows us to observe some differences between PaLM and GNU-Prolog resolution strategies. In PaLM (Fig. 21-left) all propagation steps happen after all constraints have been posted. In the PAVOT view this is clearly revealed by a single ball after all constraints have been posted.

The Discovery (Fig. 21-middle) split parallel histogram display has only three visible lines corresponding to the events (from top to bottom): awake, reduce and schedule. The number of suspend events is insignificant. We observe two principal phases visible in the reduce display and revealed by the change of declivity of the top line: one when a reduce event concerns one constraint and one variable at once (restriction of the domains after all constraints are posted, low declivity), and a second one where all reduce events are separated by several other events (awake and schedule events), i.e. cycles of reduce and schedule and awake events (stronger declivity).

The VISADJ-view (Fig. 21-right) has been commented above in Sec. 5.2. It shows another aspect of the PaLM strategy (in this view the constraints have been ordered according to the order of posting): closer constraints have more interaction.

The possibility to use several debugging tools to observe several modelling of a problem with different solvers offers richer possibilities to analyze the problem resolution.

6 Related Work

Two event-oriented tracers have already been implemented into real solvers. The first one, a tracer for Ilog Solver, provides some tracing facilities as a class of its solver library [18]. It is fully dedicated to Ilog Solver but we have shown in [7], that it can be seen as a specialisation of our generic trace schema. A few tools have been built upon this execution trace. For example, the Ilog Christmas Tree [4] processes an XML trace performed by the debugger thanks to the overriding of its virtual functions. However adapting a new tool would require a deep understanding of the solver behaviour. Moreover, the amount of data available at each event is very limited compared to the full state that our schema enables. For instance, the set of constraints and variables cannot be traversed.

The second one, a tracer for the FD solver of Sicstus Prolog, has been prototyped by Ågren et al [2]. The trace is dedicated to Sicstus: the schema comes from the platform implementation.

Both tracers provide solver-dependent data about the execution. For instance, some details about the awaken demons or about the methods used to reduce a domain. Those data could be added in the current model as specific extensions: refinement of the constraint identification system, additional attributes of generic events, or specific events inserted into the generic events. The generic kernel of the trace is preserved while enabling solver-dependent analysis.

Some debugging tools enable users to interact with the execution states. Users can act on the current state of the execution to drive the search-tree exploration (Oz Explorer [27]), to add new constraints on a partial solution (CLPGUI [14]), to recompute a former state (both) or to cancel some constraints chosen thanks to failure explanations (PaLM [19]). Such interactions are not taken into account in the generic observational semantics (only the output flow of information is considered), but it would be useful to formalize such interactions in a generic way too. We will consider such extension in a future work.

It must be observed that interactive tools essentially use online tracing and debugging. Our first experimentation used post-mortem traces from different origins. The presented approach is however suitable for both online tracing and post-mortem analysis. Indeed, the contents of the flow of information produced by a tracer can be filtered according to what is requested by the debugging tools [11, 12]. Therefore, no useless information is sent over to the tools. The tools presented in Sec. 5 are currently being adapted to support on the fly analysis too.

7 Conclusion

We have presented a generic trace schema based on a generic observational semantics which formalizes relevant aspects of finite domain constraint programming and solving. Based on this semantics, tracers can then build specialized generic traces in which most of the debugging tools can find the information they need. In this paper the specialization functions have been rigorously but informally presented. In order to facilitate the use of the generic observational semantics on new solvers it is the matter of a current work to define these functions more formally.

We have shown that at least three representative families of finite domain solvers can be observed by this semantics: constraint logic programming (Gnu-

Prolog), CSP library (Choco) and explanation based CP (PaLM)[5]. We have also shown that three different debugging tools can find in the specialized traces the whole information they require.

We would like to emphasize that there is no restriction on the number or on the size of the attributes of the generic trace. One may object that such a trace may slow down considerably any tracer. This problem can be overcome as there exist mechanisms in order to tell tracers to produce only what is needed for a given analysis. Hence the actual flow of information does not have to be larger than what a debugging tool requires [11, 12]. Such mechanism is currently implemented in Codeine. It will ease the use of the Gnu-Prolog tracer by providing a convenient programming interface to implement trace-based debugging tools.

Several tracers based on the generic trace approach have been implemented by different groups. Our experimentation with Codeine, the GNU-prolog propagation tracer, shows that such a tracer is reasonably efficient [22]. Moreover the specialized observational semantics plays the role of a specification and the implementation of the tracer is considerably facilitated [13]. On the other side, several new debugging tools (based on recent visualization techniques) have been developed independently using the generic trace as input data only. They can be used on several solvers without difficult adaptation. The generic observational semantics can be viewed as a standard for the output of solvers and input of the tools. This standard has currently the form of an XML DTD defining the syntax of the generic trace schema [15].

The generic trace schema should not be considered as definitively settled. Although first experimentation has shown its robustness, more information may be needed when new tracers and new tools will be considered. However the presented schema allows extended experimentations with many debugging tools to be started. We intend to pursue such experimentations. We also expect that this approach contributes to facilitate and to stimulate the production of propagation tracers and new debugging tools.

Acknowledgements

The authors would like to thank Narendra Jussien and Guillaume Rochart for their technical explanations about the exact behaviour of the PaLM platform, Mohammad Ghoniem and Thomas Baudel for explanations about VISADJ and Discovery respectively. We also thank anonymous referees for their many interesting and useful remarks.

References

1. A. Aggoun, D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Meier, D. Miller, S. Mudambi, B. Perez, E. van Rossum, J. Schimpf, P.A. Tsahageas, and D.H. de Villeneuve. *ECLIPSE 3.5. User Manual.* European Computer Industry Research Centre (ECRC), Munich, 1995.

[5] Another tracer is currently being developed for CHIP [5].

2. M. Agren, T. Szeredi, N. Beldiceanu, and M. Carlsson. Tracing and explaining execution of CLP(FD) programs. In A. Tessier, editor, *Proceedings of the 12th Workshop on Logic Programming Environments*, Copenhagen, Denmark, July 2002. Available as CoRR:cs.SE/0207047.
3. T. Baudel and et al. DISCOVERY reference manual, 2003. Manufactured and distributed by Ilog, `http://www2.ilog.com/preview/Discovery/`.
4. C. Bracchi, C. Gefflot, and F. Paulin. Combining propagation information and search-tree visualization using OPL studio. In A. Kusalik, M. Ducassé, and G. Puebla, editors, *Proceedings of Proceedings of the Eleventh Workshop on Logic Programming Environments (WLPE'01)*, pages 27–39, Cyprus, Décembre 2001. Cyprus University. Available as CoRR:cs.PL/0111042, `http://xxx.lanl.gov/abs/cs.PL/0111042`.
5. Cosytec. CHIP++ Version 5.2. documentation volume 6. `http://www.cosytec.com`, 1998.
6. Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03: Sixteenth International Florida Artificial Intelligence Research Society Conference*, pages 172–176, St. Augustine, Florida, USA, May 2003. AAAI press.
7. P. Deransart, M. Ducassé, and L. Langevine. A generic trace model for finite domain solvers. In Barry O'Sullivan, editor, *Proceedings of the second International Workshop on User Interaction in Constraint Satisfaction (UICS'02)*, Cornell University (USA), Aug 2002. Available at `http://www.cs.ucc.ie/~osullb/UICS-02/papers/deransart_et_al-uics02.ps`.
8. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog, The Standard; Reference Manual.* Springer Verlag, April 1996.
9. P. Deransart, M. Hermenegildo, and J. Małuszyński, editors. *Analysis and Visualization Tools for Constraint Programming.* Number 1870 in LNCS. Springer Verlag, 2000.
10. D. Diaz. GNU-Prolog, a free Prolog compiler with constraint solving over finite domains, 2003. `http://gprolog.sourceforge.net/`, Distributed under the GNU license.
11. M. Ducassé. Opium: an extendable trace analyzer for Prolog. *The Journal of Logic Programming,* special issue on Synthesis, Transformation and Analysis of Logic Programs, 39:177–223, 1999.
12. M. Ducassé and L. Langevine. Automated analysis of CLP(FD) program execution traces. In P. Stuckey, editor, *Proceedings of the International Conference on Logic Programming.* Lecture Notes in Computer Science, Springer-Verlag, July 2002. Poster. Extended version available at `http://www.irisa.fr/lande/ducasse/`.
13. M. Ducassé, L. Langevine, and P. Deransart. Rigourous design of tracers: an experiment for constraint logic programming. In Michiel Ronsse, editor, *Proc. of AADEBUG'03.* Gent University, Sep 2003. Available as CoRR:cs.SE/6309027.
14. F. Fages. CLPGUI: a generic graphical user interface for constraint logic programming over finite domains. In A. Tessier, editor, *Proceedings of the 12th Workshop on Logic Programming Environments (WLPE'02)*, Copenhagen, Aug 2002. Available as CoRR:cs.SE/0207048.
15. J.-D. Fekete and al. Proposition de format concret pour les traces générées par les solveurs de contraintes. Public deliverable d2.2.2.1, INRIA, INRIA (France), Novembre 2001. `http://contraintes.inria.fr/OADymPPaC/Public/delivrables.html`.

16. G. Arnaud. PAVOT Reference Manual, 2004. `http://contraintes.inria.fr/~arnaud/pavot/`.
17. M. Gonhiem, N. Jussien, and J.-D. Fekete. Visualizing explanations to exhibit dynamic structure in constraint problem satisfaction. In B. O'Sullivan, editor, *Proceedings of the third International Workshop on User Interaction in Constraint Satisfaction (UICS'03)*, Cork University (Ireland), Sept 2003. Available at `http://www.cs.ucc.ie/~osullb/UICS-03/`.
18. Ilog. SOLVER 5.1 reference manual, 2001. `http://www.ilog.com/`.
19. Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.
20. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, July 2002.
21. François Laburthe and the OCRE research group. *CHOCO, a Constraint Programming kernel for solving combinatorial optimization problems*, September 2001. Available at `http://www.choco-constraints`.
22. L. Langevine, P. Deransart, and M. Ducassé. A propagation tracer for GNU-Prolog: from formal definition to efficient implementation. In C. Palamidessi, editor, *Proc. of 19th International Conference on Logic Programming (ICLP 2003)*, volume 2916, pages 269–283. Springer Verlag, septembre 2003.
23. L. Langevine, P. Deransart, Mireille Ducassé, and Erwan Jahier. Tracing execution of CLP(FD) programs, a trace model and an experimental validation environment. RR 4342, INRIA, Rocquencourt (France), Décembre 2001. Also in post-ICLP'01 workshop: 11th Workshop on Logic Programming Environments.
24. Ludovic Langevine. Codeine, a propagation tracer for GNU-Prolog, 2003. `http://contraintes.inria.fr/~langevin/codeine`.
25. M. Meier. Debugging constraint programs. In U. Montanari and F. Rossi, editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, number 976 in LNCS, pages 204–221. Springer Verlag, 1995.
26. T. Müller. Practical investigation of constraints with graph views. In *Principles and Practice of Constraint Programming – CP 2000*, number 1894 in LNCS. Springer-Verlag, 2000.
27. C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In *Proceedings of the Fourteenth International Conference on Logic Programming (*ICLP *'97)*, pages 286–300, Leuven, Belgium, June 1997. The MIT Press.
28. Helmut Simonis, Abder Aggoun, Nicolas Beldiceanu, and Eric Bourreau. Complex constraint abstraction : Global constraint visualization. In P. Deransart, M. Hermenegildo, and J. Małuszyński, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, chapter 12. Springer Verlag, 2000.
29. Pascal Van Hentenryck. *Constraint satisfaction in logic programming.* The MIT Press, 1989.

Teaching Constraints through Logic Puzzles

Péter Szeredi[1,2]

[1] Dept. of Computer Science and Information Theory,
Budapest University of Technology and Economics
H-1117 Budapest, Magyar tudósok körútja 2
szeredi@cs.bme.hu
[2] IQSYS
Information Systems Ltd.
H-1135 Budapest, Csata u. 8

Abstract. The paper describes the experiences of teaching a constraint logic programming course at the Budapest University of Technology and Economics. We describe the structure of the course, the material covered, some examples, assignments, and examination tasks. Throughout the paper we show how logic puzzles can be used to illustrate constraint programming techniques.

1 Introduction

Logic puzzles are getting more and more popular. For example, in the last few years, four monthly periodicals, publishing logic puzzles only, have been established in Hungary. There are numerous web-sites offering this type of amusement. Best puzzle solvers gathered in October 2003 in the Netherlands for the 12th time to compete in the World Puzzle Championship. Puzzles from earlier Championships were published in e.g. [1, 2].

Personally, I found logic puzzles quite intriguing, and started to use them in my Prolog lectures at the Budapest University of Technology and Economics (BUTE), and later also in the lectures on constraint logic programming (CLP). This is not new, as puzzles have been used for demonstrating the capabilities of CLP from the very establishment of this discipline. For example, Pascal van Hentenryck devotes a complete section to solving puzzles in his seminal book [3].

This paper gives an account of the constraint course at BUTE, with a brief summary of the preceding Prolog course. While describing various aspects of the course, I highlight those parts where logic puzzles are used.

The paper is structured as follows. First, in Sect. 2, the prerequisite Prolog course is briefly introduced. The next two sections describe the structure of the constraint course. Section 5 presents the details of one of the case studies, in which two solvers for the Domino puzzle are developed. Section 6 describes the major assignments of the past four years, while Sect. 7 presents some examination problems. In the last two sections we discuss the lessons learned and conclude the paper. Finally, Appendix A contains the complete code of a solver for “knights, knaves, and normals” type puzzles, while Appendix B gives a detailed performance evaluation of the Domino puzzle solvers.

2 Background

The paper presents an elective constraint logic programming course for undergraduate students of informatics at BUTE[1]. It presupposes the knowledge of logic programming, which is taught in a compulsory "Declarative Programming" (DP) course in the second year. The DP course also covers functional programming, which is represented by Moscow SML, and is taught by Péter Hanák. The logic programming part, presented by Péter Szeredi, uses SICStus Prolog.

The DP course, in part due to its late introduction into the curriculum, is a lectures only course, with no laboratory exercises. It is very difficult to teach programming languages without the students doing some programming exercises. In an attempt to solve this problem, we have developed a computer tool, called ETS (Electronic Teaching aSsistant, or Elektronikus TanárSegéd in Hungarian) [4], which supports Web-based student exercising, assignment submission/evaluation, marking, etc.

During the DP course we issue 4–6 so called minor programming assignments and a single major one. The major assignment normally involves writing a solver for a logic puzzle, a task very much suited for constraint programming. Examples of such puzzles are given in Sects. 5, 6, and 7. The marks for the major assignment are based on the number of test cases solved by the student's program, as well as on the quality of the documentation. There is a so called "ladder competition" for students who have solved all the normal test cases. In this competition additional points are awarded to students who can solve the largest puzzles.

This creates some interest in constraint programming, especially when the students are told that their puzzle-solving programs can be made much faster, often by two orders of magnitude, by using CLP. Because of this, a relatively large number of students enrol in the elective constraints course immediately, in the semester following their DP course. For example, 55 students registered for the constraints course in the autumn term of 2002, of which 38 took DP in the previous semester[2].

3 The Structure of the Constraint Course

The constraint course has been presented since 1997, eight times up till 2003 (in the spring term till 2000, and in the autumn term since 2000).

There is a 90 minute lecture each week during the term, which usually lasts 14 weeks. There are no laboratory exercises for this course. To ensure that students do some actual programming, there is (and has always been) a compulsory major assignment, which can be submitted up till the end of the exam session following the term. To motivate and help the students in doing programming exercises during the term, minor assignments were introduced in the autumn term of

[1] Occasionally the course is taken by students of electrical engineering, and by postgraduates.

[2] Altogether 324 students completed the DP course in the spring 2002 semester, of which 62 got the highest grade (5), of which 27 enrolled in the constraints course.

Table 1. The layout of the course

	Topic	Time	Slides
1.	Prolog extensions relevant for CLP *Minor Assignment 1:* CLP(MiniB)	3 units	15 slides
2.	The CLP($\mathcal{X}$) scheme and CLP(R/Q)	3 units	19 slides
3.	CLP(B)	2 units	8 slides
4.	CLP(FD) *Minor Assignments 2-4, Major Assignment*	14 units	109 slides
5.	Constraint Handling Rules	2 units	11 slides
6.	Mercury	4 units	23 slides
	$\sum$	28 units	185 slides

2002. There were four of these, with a due date of approximately two weeks after issue.

The layout of the course is shown in Table 1, with approximate time schedule, and the number of (fairly dense) handout slides for each topic. The *unit* of the time schedule is 45 minutes, i.e. there are 2 units per lecture.

In addition to its main topic of constraints, the course also contains a brief overview of the Mercury language[3]. In this paper we limit our attention to the constraint part of the course, which uses SICStus Prolog [5], as the underlying implementation. We will now discuss in turn the topics 1.-5. listed in Table 1,

The first topic deals with Prolog extensions relevant for constraint programming. These include the portray hook, term- and goal expansion hooks, and, most importantly, the coroutining facilities. These features are introduced by gradually developing a simple "constraint solver" on the domain of natural numbers. This system, called CLP(MiniNat), is based on the Peano arithmetic, and allows function symbols +, -, and *, as well as the usual six relational symbols. The user interface is similar to CLP(R):

```
| ?- {X*X+Y*Y=25, X > Y}.
X = 4, Y = 3 ? ;
X = 5, Y = 0 ? ; no
```

Using goal expansion, such constraints are transformed into a sequence of calls to predicates `plus/3` and `times/3`. Both predicates block until at least one of the arguments is ground, and then execute, possibly creating choice-points.

[3] The actual title of the course is "High Efficiency Logic Programming". The title reflects the original idea, which was to split the course into two, roughly equal parts: one about efficient compilation of logic programs, exemplified by the Mercury system, and the other about applying constraint techniques for solving search problems efficiently, using constraint techniques. However, from the very start, the focus shifted to the constraint topic.

Assignment 1 is issued at the end of the first part. Here the students are asked to write a coroutining-based simple quasi-CLP system on Booleans, called CLP(MiniB).

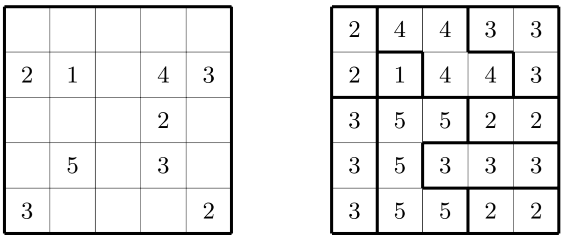
The second part starts with a brief overview of the CLP($\mathcal{X}$) scheme, introducing the notions of constraint store, primitive and non-primitive constraints. These are then exemplified by discussing the CLP(Q/R) extension, culminating in Colmerauer's Perfect Rectangle Problem [6]: tiling a rectangle with different squares. Next, a summary of the declarative and procedural semantics of the CLP scheme is given. This setup, where the theoretical issues are discussed after a concrete example of the CLP($\mathcal{X}$) scheme is shown, seems to be more easily digested by the students.

The next part presents the second instantiation of the CLP scheme: CLP(B). This is illustrated by simple examples of circuit verification, and a program playing (the user part of) the well known minesweeper game.

The fourth part, dealing with CLP(FD), is naturally the largest and is discussed in the next section. There are three minor assignments related to this part, and the major assignment is also issued here.

The constraint part of the course is concluded with a brief overview of CHR (Constraint Handling Rules). Here the biggest example is a CHR program for solving the Areas puzzle (see Fig. 1).

The Areas puzzle: A rectangular board is given with some squares specified as positive integers. Fill in all squares of the board with positive integers so that any maximal contiguous set of squares containing the same integer has the area equal to this integer (two squares are contiguous if they share a side). An example puzzle and its unique solution (with thick lines marking the area boundaries) is given below:



2	1		4	3
			2	
	5		3	
3				2

2	4	4	3	3
2	1	4	4	3
3	5	5	2	2
3	5	3	3	3
3	5	5	2	2

Fig. 1. The Areas puzzle, a CHR example

4 The CLP(FD) Part of the Course

The CLP(FD) topic takes up roughly half of the semester. Its subdivision is shown in Table 2. We start with a brief overview of the CSP approach. Next, the basics of the CLP(FD) library are introduced: the arithmetic and membership constraints, their execution process, the notion of interval and domain

Table 2. Subdivision of the CLP(FD) part of the course

	CLP(FD) subtopic	Time	Slides
4.1	CSP overview, CLP(FD) basics	3 units	23 slides
4.2	Reification, propositional constraints, labeling *Assignment 2:* Cross sums puzzle	3 units	25 slides
4.3	Combinatorial constraints *Major Assignment:* Magic Spiral puzzle	1 unit	12 slides
4.4	User defined constraints *Assignment 3:* Write a specific indexical *Assignment 4:* Write a specific global constraint	2 units	20 slides
4.5	The FDBG debugging tool	1 unit	10 slides
4.6	Case studies	4 units	19 slides

consistency, the importance of redundant constraints. This is exemplified by the classical CSP/CLP(FD) problems: map colouring, n-queens, the zebra puzzle, and the magic series.

The last example (magic series) leads to the topic of reified and propositional constraints, interval and domain entailment. An elegant illustration of the propositional constraints is a program for solving "knights, knaves and normals" puzzles of Smullyan [7], shown in Appendix A.

Although simple uses of the labeling predicates were already introduced, the full discussion of labeling comes at this point. This includes user-definable variable selection and value enumeration functions, as well as a comparison of the performance of various labeling schemes on the n-queens problem.

Having covered all the stages of constraint solving, the next minor assignment is issued. The students are required to write a program for solving a numeric crossword puzzle. Figure 2 describes the assignment and gives a sample run.

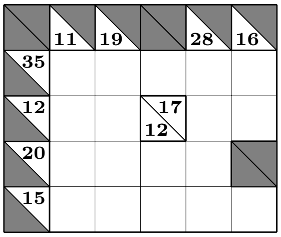
The next subtopic is an overview of combinatorial constraints provided by SICStus Prolog. This is rather dry, not much more than the manual pages, lightened up by a few small additional examples.

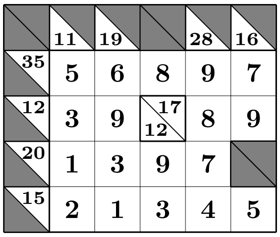
At this point the major assignment is issued. In the last three years, this was exactly the same assignment as the one issued for the Declarative Programming course in the preceding semester (see Sect. 6 for examples).

The fourth subtopic of the CLP(FD) part is about user-definable constraints. Both global constraints and FD-predicates (indexicals) are covered, including the reifiable indexicals. The last two minor assignments are about writing such constraints, as shown in Fig. 3.

When students submit an assignment, the ETS teaching support tool runs it on several predefined test-cases, and sends the results back to the student. I had to repeatedly extend the set of test cases for the FD-predicate assignment, which seems to be the most difficult one, as it was not exhaustive, and thus

The Cross Sums puzzle: Fill in a "numeric" crossword puzzle, with integers from the $[1, Max]$ interval. All integers within a "word" have to be different, and the sum of these is given as the "definition" of each word. An example and its solution ($Max = 9$):





Assignment 2: Write a CLP(FD) program for solving the Cross Sums puzzle. Sample run:

```
| ?- T = [x\x, 11\x,19\x, x\x,28\x,16\x],
        [x\35,   _,   _,   _,   _,   _],
        [x\12,   _,   _,12\17,  _,   _],
        [x\20,   _,   _,   _,   _, x\x],
        [x\15,   _,   _,   _,   _,   _]],
     cross_sums(T, 9).

     T = [x\x, 11\x,19\x, x\x,28\x,16\x],
        [x\35,   5,   6,   8,   9,   7],
        [x\12,   3,   9,12\17,  8,   9],
        [x\20,   1,   3,   9,   7, x\x],
        [x\15,   2,   1,   3,   4,   5]] ? ;
no
| ?-
```

Fig. 2. The Cross Sums puzzle

Assignment 3: Write an FD-predicate `'z>max(x,y)'(X, Y, Z)` which implements a domain-consistent constraint, with the meaning equivalent to `Z #> max(X,Y)`. Write all four clauses of the FD-predicate.

Assignment 4: Write a global constraint `max_lt(L, Z)`, where `L` is list of FD variables, and `Z` is an FD variable. The meaning of the constraint: the maximum of `L` is less than `Z`, Make the global constraint efficient, avoid re-scanning irrelevant variables, by using a state. For example, the following goal should run in linear time with respect to `N`:

```
| ?- N = 500, length(L, N), domain(L, -5, 0), X in 0..N,
     max_lt([X|L], Z), X#>0, X#>1, ..., X#>N-1.
```

Fig. 3. Minor assignments for writing user-defined constraints

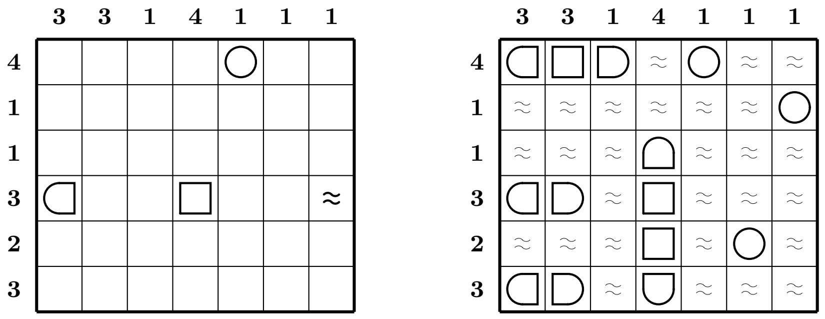


Fig. 4. A Battleship puzzle with its solution

incorrect student solutions got accepted. Finally, I developed, and handed out to students, a simple tool for exhaustively checking the semantics of an FD-predicate against a specification in Prolog. The specification is a Prolog goal for *testing* the constraint with ground arguments. The semantic check can be run automatically, if some directives are inserted in front of the FD-predicate. In the case of Assignment 3 (Fig. 3) these may look like this:

```
:- fd_pred_semantics('z>max(x,y)'(X,Y,Z), Z>max(X,Y)).
:- fd_test_range(1, 2).      % Try all combinations in the range 1..2
```

The next subtopic within CLP(FD) is debugging, as the students are expected now to start developing their major assignments. A brief overview of the SICStus FDBG finite domain debugging library [8] is presented here. An interesting point is that this library has been developed by two former students of this course.

The CLP(FD) part is concluded by three case studies. First the Perfect Square Problem from [9] is presented. This gives an opportunity to discuss disjunctive constraints and special labeling techniques. Performance results are presented for several solution variants, including those using the SICStus Prolog library predicates `cumulative` and `disjoint2`.

The next two case studies are major assignments from earlier years: the Battleship and the Domino puzzles. The latter is discussed in the next section.

The Battleship puzzle involves placing ships of different lengths on a rectangular board (all ships are one square wide). The ships can be placed horizontally or vertically, and they can be of different colours. For each length and colour we are given the number of battleships to be placed. We are also given, for each colour, the number of battleship pieces in each row and column. The battleships cannot touch each other, not even diagonally. Furthermore, for certain squares of the board it is specified that they contain a certain kind of battleship piece, or sea. Ships of length one have a circle shape, while longer ships have rounded off bow and stern pieces, with square mid-pieces in between.

Figure 4 shows an example of a Battleship puzzle, with a single colour, together with its solution. The numbers shown above the columns and in front of the rows are the counts of battleship pieces. On the board, three ship pieces are given, and one field is marked as sea. In addition to the board on the left hand side, the puzzle solver is given the ship counts: length 4: 1 ship, length 3: 1 ship, length 2: 2 ships, length 1: 3 ships.

The Battleships case study (program, test-data, results) can be downloaded from [10].

5 A Case Study: The Domino Puzzle

The Domino puzzle was issued as the major assignment in spring 2000. Since 2001, it has been used as a case study, showing the steps and techniques needed to solve a relatively complex problem using CLP.

5.1 The Puzzle

A rectangular board is tiled with the full set of dominoes with up to d dots on each half domino. The set of all such dominoes can be described as:

$$D_d = \{\langle i, j\rangle | 0 \leq i \leq j \leq d\}$$

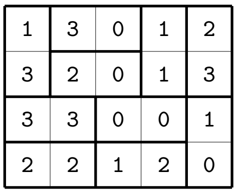
For each square of the board we are told the number of dots on the half domino there, but we do not know the domino boundaries. The task is to find out the tiling, i.e. to reconstruct the domino boundaries.

Figure 5 shows a sample board, its solution, and the Prolog format of these.

The problem:

1	3	0	1	2
3	2	0	1	3
3	3	0	0	1
2	2	1	2	0

The (unique) solution:



The Prolog description:

```
[[1,  3,  0,  1,  2],
 [3,  2,  0,  1,  3],
 [3,  3,  0,  0,  1],
 [2,  2,  1,  2,  0]]
```

The solution in Prolog:

```
[[n,  w,  e,  n,  n],
 [s,  w,  e,  s,  s],
 [w,  e,  w,  e,  n],
 [w,  e,  w,  e,  s]]
```

Fig. 5. A sample Domino problem for $d = 3$

The task is to write a `domino/2` predicate, which takes, as its first argument, a board represented by a matrix of values from $[0, d]$, and returns, in the second argument, a matrix of the same size. The elements of the output matrix are compass point abbreviations `n`, `w`, `s`, or `e`, indicating that the square in question is respectively a northern, western, southern, or eastern half of a domino.

In the case study two alternative solutions are presented: one based on the *compass* model, and another one using the *border* model. An independent Eclipse solution to this puzzle, found in [11], uses the latter model.

5.2 The Compass Model

This model follows naturally from the expected output format: each square of the board is assigned a *compass* variable, which specifies the compass point of the half domino it is covered with. The compass variables are named $CV_{yx}, 1 \leq y \leq max_y, 1 \leq x \leq max_x$ (where max_y and max_x are the number of rows and columns of the board), their domain is an arbitrary numeric encoding, say n, w, s, e, of the four compass points.

With this representation, it is easy to ensure that the tiling is consistent: if a square is "northern", its neighbour to the south has to be "southern", etc. However it is difficult to guarantee that each domino of the set is used only once.

Therefore we introduce another, redundant set of variables. For each domino $\langle i, j\rangle \in D_d$, we set up a *domino* variable DV_{ij} specifying the position of this domino on the board. This automatically ensures that each domino is used exactly once.

The domino positions can be described e.g. by suitably encoding the triple $\langle row, column, dir\rangle$, where the coordinates are those of the northern/western half of the domino, and *dir* specifies its direction (vertical or horizontal). If a domino $\langle i, j\rangle$ can be placed on k positions, then the domain of D_{ij} will be $[1, k]$. The mapping from this domain to the triples is needed only while posting the constraints, and so it can be kept in a separate Prolog data structure.

For example, the $\langle 0, 2\rangle$ domino of Fig. 5 can be placed on the following board positions: $\langle 2, 2, horizontal\rangle$, $\langle 3, 4, vertical\rangle$ and $\langle 4, 4, horizontal\rangle$. Therefore the domain of D_{02} will be $\{1, 2, 3\}$, describing these three placings.

The constraints of the compass model are the following:

Neighbourship constraints: For each pair of neighbouring squares on the board we state that their compass variables have to be consistent, e.g. $CV_{14} = n \Leftrightarrow CV_{24} = s$, $CV_{14} = w \Leftrightarrow CV_{15} = e$, etc.

Placement constraints: For each domino variable, and for each of its possible values, we state that the given horizontal (vertical) placement holds iff the compass variable of the square specified by this placement is a western (northern) half of a domino. For example, the $\langle 0, 2\rangle$ domino of Fig. 5 gives rise to the following constraints: $DV_{02} = 1 \Leftrightarrow CV_{22} = w$, $DV_{02} = 2 \Leftrightarrow CV_{34} = n$, $DV_{02} = 3 \Leftrightarrow CV_{44} = w$.

Note that both constraint types are of form $X = c \Leftrightarrow Y = d$, where c and d are constants. Three implementations for this constraint are presented. The first variant is the trivial formulation using reification and propositional constraints.

The second uses an FD-predicate implementing the constraint $X = c \Rightarrow Y = d$, and calls it twice, to ensure equivalence. Third, the whole equivalence is coded as a single FD-predicate. The two FD-predicates are shown in Fig. 6. All solutions have the same pruning behaviour. Our measurements (see Table 4 in Appendix B) show that the second is the fastest variant. This involves four indexicals, as opposed to the the third, single FD-predicate implementation, which has only two. However, in the latter case the indexicals are more complex, and wake up unnecessarily often, hence the inferior performance of this solution.

```
'x=c=>y=d'(X, C, Y, D) +:
        X in (dom(Y) /\ {D}) ? (inf..sup) \/ \({C}),
        Y in ({X} /\  \({C})) ? (inf..sup) \/ {D}.

'x=c<=>y=d'(X, C, Y, D) +:
        X in ((dom(Y) /\ {D}) ? (inf..sup) \/ \({C})) /\
             ((dom(Y) /\ \({D})) ? (inf..sup) \/ {C}),
        Y in ((dom(X) /\ {C}) ? (inf..sup) \/ \({D})) /\
             ((dom(X) /\ \({C})) ? (inf..sup) \/ {D}).
```

Fig. 6. FD-predicates for implementing equivalence constraints

5.3 The Border Model

In this model we assign a variable to borders (line segments) between the squares of the board. Such a *border* variable is 1, if the given segment is a centerline of a domino, otherwise it is 0. Let us denote by E_{yx} (S_{yx}) the variables corresponding to the eastern (southern) borders of the square (y, x) on the board ($1 \leq y \leq max_y, 1 \leq x \leq max_x$) [4].

Analogously to the compass model, we have two types of constraints:

Neighbourship constraints: For each square on the board we state that *exactly one* of the four line segments bordering it will be a domino centerline, i.e. the sum of the corresponding variables is 1. For example, for the square $(2, 4)$ the constraint is: $S_{14} + E_{23} + S_{24} + E_{24} = 1$.

Placement constraints: For each domino, consider all its possible placements. We state that exactly one of these placements has to be selected, i.e. from amongst the line segments in the centre of these placements, exactly one will be a domino centerline. For example, the $\langle 0, 2\rangle$ domino of Fig. 5 gives rise to the following constraint: $E_{22} + S_{34} + E_{44} = 1$.

Note that again both constraint types are of the same form: the sum of some 0-1 variables is 1. Two implementations are evaluated for the $\sum X_i = 1$ constraint: one using the `sum/3` global constraint of SICStus Prolog, the other using

[4] Note that in this set of variables there are some, which correspond to line segments on the outer border of the board (e.g. $S_{max_y x}$), these are assigned a constant 0 value. To cover the northern and the western border of the board, we use similarly constant 0 valued variables S_{0x} and E_{y0}.

indexicals. For the latter we make use of the fact that the SICStus `clpfd` library is capable of compiling linear arithmetic constraints to indexicals. For example, an FD-predicate implementing a three-way sum constraint can be specified as `sum3(A,B,C) +: A+B+C #= 1.` Measurements show that the FD-predicate is faster up to 5 summands, and slower above that.

5.4 Search

In the compass model we have a choice of which variable-set to label: the compass variables, or the domino variables. Furthermore, in both models, we can explore different variable selection schemes. However, for large test cases to be solved in acceptable time, we have to apply an additional technique, *shaving*.

Shaving [12] is a straightforward, indirect proof scheme, which seems to work well for logic puzzles. It involves tentatively posting a constraint: if this leads to failure, then the negation of the constraint is known to hold. Otherwise the constraint is withdrawn, using the “negation by failure” scheme. The following piece of Prolog code tries out if setting `X` to `Value` leads to failure, and excludes the given value from the domain of `X` if this happens.

```
shave(X, Value) :-
        (   \+ X #= Value -> X #\= Value
        ;   true
        ).
```

Such shaving operations are normally performed during labeling. For example, in the Battleship puzzle case study, shaving was used to exclude a specific value (the one corresponding to “sea”) from the domain of the FD variables describing the squares of the board. This process was performed relatively rarely, before labeling the placement of ships of each colour. Experiments showed that it is worthwhile to do shaving, but it does not pay off to do repetitive shaving (i.e. repeat the shaving, until no more domains are pruned).

A slightly more general shaving scheme is presented in the Domino case study. We still try to prove inconsistency by setting an FD variable to a concrete value. However, here one can specify multiple values to be tried in shaving, one after the other. The frequency of doing (non-repetitive) shaving can also be prescribed: a shaving scan can be requested before labeling every kth variable.

In the compass model, a shaving step involves scanning all the compass variables. Out of the two opposite compass values (e.g. northern and southern) it is enough to try at most one, because the neighbourship constraints will force the shaving of the other value. Also, shaving all compass variables with two non-opposite values will result in all domino variables being substituted with all their possible values, because of the placement constraints. Consequently, there is no need to shave the domino variables. In the compass model we thus shave the compass variables only, trying out the $[n, w]$ and $[n]$ value sets.

In the border model we explore shaving variants using value sets $[0]$, $[1]$, and $[0, 1]$.

The case study is concluded with a relatively detailed performance evaluation, the essence of which is shown in Appendix B.

6 Major Assignments

In 1997 and 1998 the major assignments were the same: writing a Nonogram solver[5]. The major assignments of the 1999 and 2000 spring semesters, the Battleship and Domino puzzles, later became part of the course as case studies. These were discussed in the previous sections. In this section we describe the problems set as major assignments in (the autumn semesters of) years 2000–2003. As opposed to the case studies, here we do not publish any details of the solution, so that the problems can be re-used in other constraint courses.

6.1 The Countries Puzzle

The Countries puzzle was set as the major assignment of the constraint course in the autumn semester of 2000.

The Puzzle. A rectangular board of size $n * m$ is given. Certain fields in the board contain numbers, these represent the capitals of different countries. Each country can only occupy contiguous fields in the row and/or in the column of the capital. This means that each country consists of the capital, plus $I_n \geq 0$ adjacent fields northwards, $I_w \geq 0$ adjacent fields westwards, $I_s \geq 0$ adjacent fields southwards, and $I_e \geq 0$ adjacent fields eastwards. The number $I \geq 0$ given on the board is the number of fields occupied by the given country, in addition to the capital, i.e. $I = I_n + I_w + I_s + I_e$.

The task is to determine, how much each country extends in each of the four main compass directions[6].

Figure 7 shows an instance of the puzzle for $n = 4, m = 5$, together with its (single) solution, showing the country boundaries.

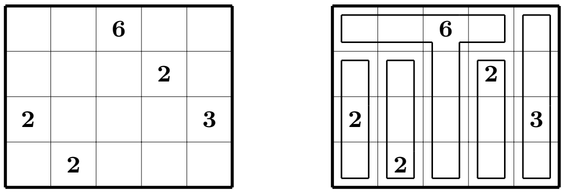


Fig. 7. A Countries puzzle and its solution

[5] Nonongrams are known by many other names, see e.g. [13]. A similar student project, for developing a nonogram solver using constraint techniques, is described on page [14].

[6] In Hungarian, this puzzle is called the compass point *(szélrózsa)* puzzle.

The Prolog Representation. A Countries puzzle is described by a Prolog structure of form `c(`*n*`,`*m*`,`*Capitals*`)`, where *Capitals* is a list of capitals, each represented by a structure `c(`*Row*`,`*Column*`,`*CountrySize*`)`. A solution is a list which, for each capital, as listed in *Capitals*, gives the extent of its country in the four compass directions. Each element of the solution list is a structure of form `e(`*North*`,`*West*`,`*South*`,`*East*`)`.

Below is a sample run of the Countries program, for the test case of Fig. 7.

```
| ?- countries(c(4, 5,
                 [c(1,3,6),c(2,4,2),c(3,1,2),c(3,5,3),c(4,2,2)]), S).
S = [e(0,2,3,1),e(0,0,2,0),e(1,0,1,0),e(2,0,1,0),e(2,0,0,0)] ? ;
no
```

6.2 The Tents Puzzle

The Tents puzzle was the major assignment for the declarative programming course in spring 2001, and for the constraint course in autumn 2001.

The Puzzle. A rectangular board of size $n * m$ is given. Certain fields in the board contain *trees.* The task is to tie a single *tent* to each tree. The tent should be placed on a field next to its tree, either to the south, west, north, or east, but not diagonally. No tents touch each other, not even diagonally. As an additional constraint, the number of tents is specified for some rows and columns.

Figure 8 shows an instance of the puzzle for $n = m = 5$, together with its single solution, where the trees are denoted by the Υ symbol, and the tents by Δ. The numbers in front of the board and above it are the tent counts for the rows and the columns.

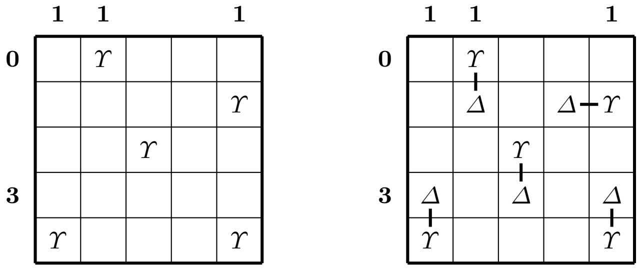


Fig. 8. A Tents puzzle and its solution

The Prolog Representation. The Prolog structure `t(`*RCs*`,`*CCs*`,`*Ts*`)` is used to describe a Tents puzzle, where *RCs* is a list of length n, containing the tent-counts of the rows. Similarly, the list *CCs* of length m gives the tent-counts of the columns. Finally, *Ts* is a list specifying the coordinates of the trees in the form of *Row*-*Column* pairs. In the first two lists, `-1` stands for an unspecified count. A solution is a list which, for each tree as listed in *Ts*, gives the direction in which its tent should be placed. The latter is specified as one of the letters `n`, `w`, `s`, or `e`. Below is a sample run of the tents program:

```
| ?- tents(t([0,-1,-1,3,-1],[1,1,-1,-1,1],[1-2,2-5,3-3,5-1,5-5]), Dirs).
Dirs = [s,w,s,n,n] ? ;
no
```

6.3 The Magic Spiral Puzzle

The Magic Spiral puzzle was set as the major assignment for the constraint course in the 2002 autumn term, having served the same role in the spring term for the DP course.

The Puzzle. In this puzzle a square board of $n * n$ fields is given. The task is to place integer numbers, chosen from the range $[1..m]$, $m \leq n$, on certain fields of the board, so that the following conditions hold:

1. in each row and each column all integers in $[1, m]$ occur exactly once, and there are $n - m$ empty fields;
2. along the spiral starting from the top left corner, the integers follow the pattern $1, 2, \ldots m, 1, 2, \ldots, m, \ldots$ (number m is called the period of the spiral).

Initially, some numbers are already placed on the board.

Figure 9 shows an instance of the puzzle for $n = 7$ and $m = 4$, as well as its (single) solution.

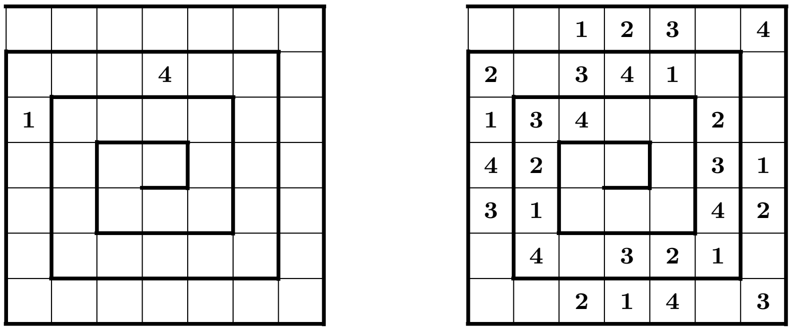


Fig. 9. A Magic Spiral puzzle and its solution

The Prolog Representation. A Magic Spiral puzzle is described by a structure of form `spiral(`n`,`m`,`*Ps*`)`, where *Ps* is a list of `i(`*Row*`,`*Column*`,`*Value*`)` triplets specifying the known elements on the board. A solution is simply a matrix represented as a list of lists of integers, where 0 means an empty field, while other integers correspond to the values on the board. Here is a sample run of the spiral program:

```
| ?- magic_spiral(spiral(7,4,[i(2,4,4),i(3,1,1)]), SpTable).
SpTable = [[0,0,1,2,3,0,4],
           [2,0,3,4,1,0,0],
           [1,3,4,0,0,2,0],
           [4,2,0,0,0,3,1],
           [3,1,0,0,0,4,2],
           [0,4,0,3,2,1,0],
           [0,0,2,1,4,0,3]] ? ;
no
```

6.4 The Snake Puzzle

The Snake puzzle was used in the spring and autumn terms of 2003 for the DP and the constraints course, respectively.

The Puzzle. A rectangular board of size $n * m$ is given. The task is to mark certain fields on the board as belonging to the *snake*, observing the following conditions:

1. The snake consists of a sequence of neighbouring fields (i.e. fields sharing a side).
2. The position of the snake head (the first field) and of the tail (last field) is given.
3. The snake can touch itself only diagonally, i.e. no snake field can have more than two snake field neighbours, but it can share vertices with other snake fields.
4. Certain fields of the table contain a number.
   (a) The snake cannot pass through fields containing a number.
   (b) If a field contains number c, then among the eight side and diagonal neighbours of this field there are exactly c which contain a snake piece. (This condition is similar to the one used in the popular minesweeper game.)

The left side of Fig. 10 shows an instance of the Snake puzzle for $n = m = 5$, with the snake head/tail marked as **H** and **T**. In the right hand side of the figure the unique solution is given, marking the fields of the snake with a filled circle.

The Prolog Representation. A Snake puzzle is represented by a structure of form `s(`*`Size,Head,Tail,RefPoints`*`)`. The first three arguments are all of form *`Row-Column`* and denote the board size, the position of the snake head and that of the snake tail. The *`RefPoints`* argument contains the list of reference points, describing the board positions containing a number. A reference point is represented by the structure `p(`*`R,C,N`*`)`, indicating that number *`N`* is placed in row *`R`*, column *`C`* of the board.

The solution of the puzzle, i.e. the snake, is represented by a list of neighbouring snake positions, starting with the head. Again, each position is given in the *`Row-Column`* format.

H		**T**		
		5		
4				
			3	

●		●	●	
●	●	**5**	●	●
4	●			●
	●	●	●	●
			3	

Fig. 10. A Snake puzzle and its solution

Below is a sample run of the a program solving the Snake puzzle, for the example of Fig. 10.

```
| ?- snake(s(5-5,1-1,1-3,[p(2,3,5),p(3,1,4),p(5,4,3)]), Snake).
Snake = [1-1,2-1,2-2,3-2,4-2,4-3,4-4,4-5,3-5,2-5,2-4,1-4,1-3] ? ;
no
```

6.5 Generating Puzzles

When making assignments, it is not enough to design a suitable project, one also has to provide appropriate test data, on which the student programs can be tested. In our case this means finding puzzles of the given type. One has to use a sequence of test puzzles of increasing difficulty, so that students can compete how far they can get in solving these.

The notion of difficulty cannot be precisely defined, but certain factors influencing the difficulty can easily be identified. For example, the puzzle normally becomes more difficult if its size increases, or if the number of the clues decreases.

In earlier years we used puzzles published in periodicals, with some variations. This was the case, e.g. for the Battleship assignment, where we started from a published puzzle and made it more difficult by taking away clues. In the last few years we used constraint techniques for generating puzzles, for both the declarative programming and the constraints course.

Generally, puzzles are built "backwards", in the following phases:

1. Build a solved puzzle.
2. Construct a puzzle from the solution,
3. Tune the puzzle difficulty.

Let us illustrate this on the generator program for the Magic Spiral puzzle. The generator is supplied the following parameters: the puzzle size (n), the period (m) and a difficulty grade, to be explained below. The generator is built around a solver for the puzzle, which incorporates a randomising labeling scheme.

First, the solver is invoked with the n, m parameters, and no integers placed on the board. By using randomised labeling, we arrive at a random magic spiral. For larger puzzles, it became important to limit the time spent in this process. In case of time-out, we restart the generation a few times, before giving up.

In the case of the Magic Spiral puzzle, the solution itself can be viewed as a (trivial) puzzle. We then build a proper puzzle, by taking away clues, i.e. numbers from the board. We have chosen to keep the "unique solution" property of the puzzle as long as possible.

The transformation of the puzzle is thus the following: we omit a randomly selected number from the spiral, and check, again using the solver, that the resulting problem has a single solution[7]. If the deletion leads to multiple solutions, we try other deletions. If none of the deletions is suitable, we arrive at a puzzle with a (locally) minimal set of clues, which still has the "unique solution" property.

The obtained minimal unique puzzle is treated as having a fixed difficulty grade, say grade 0. If this grade was requested, then the minimal unique puzzle is returned. If the difficulty grade is negative, we put back some of the clues, again randomly selected, onto the board. The lower the difficulty grade, the more numbers are put back.

On the other hand, if the difficulty grade is positive, then this is taken as a request to generate a non-unique puzzle, by taking away further clues from the board. Here we have adopted the principle that higher difficulty grades correspond to more solutions: for the difficulty grade g we keep taking away clues until a puzzle with at least g solutions is found. However, we exclude puzzles with too many, say more than 20, solutions.

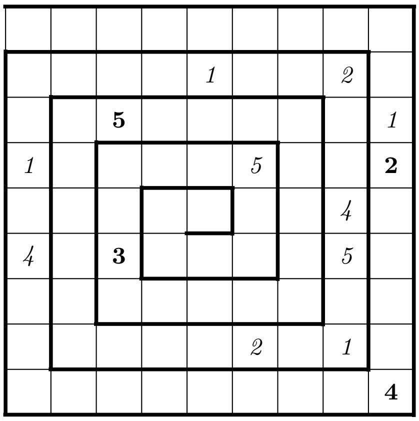


Fig. 11. A generated Magic Spiral puzzle, period 5, difficulty -5

Figure 11 shows a Magic Spiral puzzle, of difficulty grade -5, generated by the procedure described above. The numbers shown in boldface constitute the

[7] Alternatively, we may insist that the resulting problem is solvable without labeling. This latter scheme, which results in easier puzzles, was used in generating the puzzle of Fig. 11.

minimal unique puzzle, while the ones in italic are redundant clues, added to make the puzzle easier.

The puzzle in Fig. 9 has also been generated by the above procedure, using difficulty grade 0.

A similar puzzle generator was created for the Snake puzzle. Here again, we first generate a random snake. Next, we produce a trivial puzzle by inserting numbers in all non-snake fields of the board, and keep on taking away these clues until a "minimal unique" puzzle is found, etc. However, for large boards (say, above 50x50), the random snake generation using the solver proved to be infeasible. Instead, we resorted to using a simple limited backtracking search. Another speed improvement was achieved by first taking away clues in larger chunks, and halving the chunk size, when a non-unique puzzle instance was hit.

The outlined puzzle generation scheme seems to be applicable for most puzzles with a variable number of clues. However, it requires a fairly good solver to be built first. There may also be a concern that this approach results in puzzles which are biased towards the specific solver used.

7 Examination Problems

The exams are in writing, with some verbal interaction. The students are asked to solve two simpler problems selected from the following four topics: CLP(Q), CLP(B), CHR, Mercury. Next, they are asked to write an indexical and a global constraint, similar to ones given as minor assignments shown in Fig. 3. Finally, they have to solve a simple CLP(FD) problem, usually involving reification.

In the past years there were some puzzle related problems issued in this last category.

The first such task, used in 1999, is related to the Nonogram puzzle. The students were asked to write a predicate constraining a list of 0-1 variables to contain a specified sequence of blocks of 1's. This is the base constraint to be posted for each row and column of a nonogram puzzle.

The task was thus to write the following predicate:
`blocks(`*`Bits`*`, `*`N`*`, `*`Lengths`*`)`: *N* is a given integer, *Lengths* is a given list of integers, and *Bits* is a list of *N* 0..1 valued constraint variables. The list *Lengths* specifies the lengths of the blocks (maximal contiguous sequences of 1's) occurring in *Bits*. For example, the list `[0,1,1,0,1,1,1]` contains two blocks and their lengths are `[2,3]`. Examples:

```
| ?- blocks(Bits, 7, [2,3]).
        Bits = [_A,1,_B,_C,1,1,_D],
        _A in 0..1, _B in 0..1, _C in 0..1, _D in 0..1 ? ; no

| ?- blocks(Bits, 7, [2,3]), Bits=[0|_].
        Bits = [0,1,1,0,1,1,1] ? ; no
```

This task proved to be a bit too difficult, but with some help most of students could solve it. The help involved proposing some auxiliary variables: for each element of the ***Length*** list introduce a variable S_i, which is the starting index of block i. Subsequently for each block i and each bit-vector position j define a Boolean variable B_{ij} which states whether block i covers position j [8].

The second exam task, from spring 2000, is linked to the Skyscrapers puzzle.

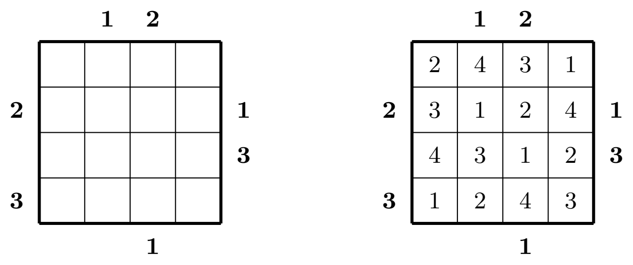


Fig. 12. A Skyscrapers puzzle and its solution

The Skyscrapers Puzzle. A permutation of integers from the `[1,`*`n`*`]` interval is to placed in each row and column of a square board, where n is the size of the board. Each field of the board represents a skyscraper, the number in the field is the height of the building. There are some numbers shown by the side of the square, they indicate how many skyscrapers are *visible* from these directions. A skyscraper is visible, if all buildings between it and the viewer are of smaller height. In Fig. 12 a sample puzzle is shown with its solution.

The students were given the above puzzle description and were then asked to write a CLP(FD) predicate which implements the following constraint, without using speculative disjunction (i.e. without creating choice-points).

`visible(`***`List`***`, `***`Length`***`, `***`NVisible`***`)`: ***List*** is a permutation of integers between 1 and ***Length***, in which the number of elements visible from the front is the given integer ***NVisible***. An element of the list is visible from the front, if it is greater than all elements preceding it. Examples:

```
| ?- visible(L,3,3).
                 L = [1,2,3] ? ;  no
| ?- visible(L,3,2), labeling([], L).
                 L = [1,3,2] ? ;
                 L = [2,1,3] ? ;
                 L = [2,3,1] ? ;
                 no
```

The third exam task to be presented is related to the End View puzzle.

[8] The constraint linking the S_i and B_{ij} variables can be stated as:

S_i `in` $(j - l_i + 1)$ `..` j `#<=>` B_{ij}

where l_i is the i^{th} element of the ***Length*** lengths list. The j^{th} element of the ***Bits*** bit-vector, B_j, can then be defined through the constraint $B_j = \sum_i B_{ij}$.

The End View Puzzle. A square board of size $n * n$ is given. Numbers from the range $[1, m]$, $m \leq n$ are to be placed on the board, so that all integers in $[1, m]$ occur exactly once in each row and each column (and there are $n - m$ empty fields in each of these). As clues, numbers are placed outside the board, prescribing the first number seen from the given direction[9].

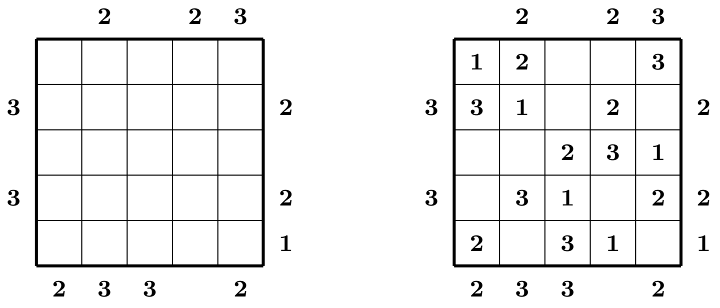


Fig. 13. An End View puzzle and its solution

Figure 13 shows a sample puzzle of this kind, for $n = 5, m = 3$, together with its unique solution. Here, for example, the number 2 placed below the fifth column prescribes that the last number placed in that column is 2, while the number 3 placed in front of row 4 requires that this number is the first in that row.

The following exam task, first set in 2001, formulates the "end view constraint" of this puzzle, assuming that empty spaces are represented by 0 values.

Write a `clpfd` constraint `firstpos(`*`List, Number`*`)`, where *`List`* contains non-negative integers, and its first positive element is equal to *`Number`*. *`Number`* is a given integer, while *`List`* is a list of constraint variables or integers. Avoid speculative disjunctions. Examples:

```
| ?- firstpos([X], 9).
                X = [9] ? ; no
| ?- firstpos([0,X,2], 1).
                X = 1 ? ; no
| ?- firstpos([A,X,2,4,3], 2), A #=< 0.
                A = 0, X in  {0}\/{2} ? ; no
```

The difficulty of this task seems to lie in the fact, that except for the first positive element, the rest of the list is not constrained. At the same time the position of the first positive element is not known. Some students needed a hint: introduce a Boolean variable for each list position, reflecting the fact that all elements before this position are zero.

[9] This puzzle is usually presented with letters A, B, C, D, etc., instead of numbers 1, 2, 3, 4, ...

8 Discussion

This section presents a brief discussion of some general issues related to the topic of this paper.

The Course. The constraint course seems to be fairly successful. In spite of the competition with less-demanding elective courses, a large number of students enrol in the course. There is a relatively large drop-out rate: in 2002 only 29 students completed the course, out of the 55 who had enrolled in it. The biggest obstacle seems to be the compulsory major assignment, which requires a considerable amount of work to complete. On the other hand, those who do submit the assignment, get very good grades: 24 students got the highest grade, 5, and only five got the next highest, 4.

The course focuses on the practical programming side of constraints, the theoretical background of CLP is not treated in detail. However, the basic notions, such as the primitive and non-primitive constraints, the constraint store, and its operations, are reiterated for each specific constraint library. This strengthens the students' understanding of the common basis of the constraint packages covered.

The discussion of the finite domain library is the most elaborate, covering a whole spectrum, from simple arithmetic constraints to user defined ones. The weakest part is on combinatorial constraints, perhaps the discussion of these could be limited to the ones actually used in the case studies. The students report that the topic of FD predicates poses the biggest challenge, but the exams prove that practically all students become capable of working with indexicals.

The application areas of constraint programming are just briefly mentioned at the beginning of the course. As one of the reviewers pointed out, a possible improvement would be to use one of the case study slots for discussing a prototype of a real-life constraint application.

The Assignments. The recently introduced minor assignments were quite popular, they gave the opportunity to practice some of the techniques described in the lectures. Students were given extra points for correct assignments, half the points were awarded even to students with late submissions.

The major assignment, which involves student competition, seems to give good motivation. Students compete not only against each other, but they would also like to beat the teacher's solution, which does happen in some cases. Usually, there is quite a bit of traffic on the course mailing list as the submission deadline approaches. The progress of some students can be seen from the questions they pose: first they write just the necessary constraints, then introduce some redundant ones, and finally try various labeling schemes and shaving.

Automated testing of assignments was found very useful. Students can submit their solutions any number of times, only the last submission is marked. The testing tool provides almost immediate feedback, at least for the minor assignments. Regarding the major assignment, where testing takes longer, we got some suggestions to improve the testing interface. For example, students asked for an option with which they can specify which test cases to run.

Logic Puzzles. Tasks involving puzzles seem to be very good means for teaching constraints. Simpler puzzles are attractive because of the very concise, logical solutions, see e.g. the knights and knaves program in Appendix A. More complex puzzles give a good opportunity to present the different phases of constraint solving: modelling, establishing constraints, enumeration of solutions. Within each phase a lot of relevant issues can be discussed, as shown in the Domino case study (Sect. 5). Students then have to go through this process themselves, when solving the major assignment.

With logic puzzles, one can try solving them by hand. Since humans rarely do backtracking search, the human process of solution often gives clues how to deduce more information, avoiding unnecessary search.

It can be argued that our course setup, with a lot of puzzles, may give the students a false impression, namely that constraint programming is only good for solving non-practical problems. Although this danger is there, I do not believe it to be a serious problem. This is demonstrated by the fact that students do recognise relevant search problems in their subsequent research and thesis work, and use CLP techniques for solving these.

Future Work. During the past decade we have accumulated quite a few constraint problems, which were formulated during the declarative programming and constraints courses. We also have numerous student solutions for these problems, in plain Prolog, Prolog with CLP extensions, as well as in SML. It would be very interesting to process the student solutions and try to find out which techniques are most often used, which are avoided, etc., giving feedback on how to improve the courses.

It would be nice to extend the local competition in solving these problems to a wider circle, both in time and in space. The former would mean that students can submit solutions to problems set in earlier terms, while the latter would enlarge the audience to any interested party outside our university/course. Obviously, the automatic assignment testing facilities of the web-based ETS system will have to be extended to support this.

9 Conclusions

Various puzzle competitions and periodicals publishing logic puzzles seem to offer an inexhaustible supply of interesting and scalable constraint problems. We have successfully used these problems in attracting and directing students' attention first towards declarative programming, and subsequently towards CLP.

In this paper we have presented an overview of the constraint course at Budapest University of Technology and Economics. We argued that logic puzzles of various kind and difficulty can be successfully used in various roles during the course: programming examples, assignments, case studies, exam tasks, etc.

We hope that the material presented here can be re-used in other constraint courses.

Acknowledgements

I am grateful to Tamás Benkő, for the numerous fruitful discussions on how to teach constraints and for developing the material for the Mercury part. Mats Carlsson was always at hand when I had problems with understanding CLP. He also kindly let me use some material from his CLP course at Uppsala University. Most major assignments presented here were originally developed for the DP course, together with Péter Hanák, Tamás Benkő, Dávid Hanák, and the student demonstrators. Dávid Hanák and Tamás Szeredi, the developers of the FDBG debugging library, produced the first version of the material describing this tool. The comments of the anonymous referees have helped a lot in improving the paper.

Thanks are also due to all the students of the course, who patiently let me learn the secrets of constraint programming while I have been pretending to teach them.

References

1. Shortz, W., ed.: Brain Twisters from the First World Puzzle Championships. Times Books (1993)
2. Shortz, W., Baxter, N., eds.: World-Class Puzzles from the World Puzzle Championships. Times Books (2001)
3. Van Hentenryck, P.: Constraint Satisfation in Logic programming. The MIT Press (1989)
4. Hanák, D., Benkő, T., Hanák, P., Szeredi, P.: Computer aided exercising in Prolog and SML. In: Proceedings of the Workshop on Functional and Declarative Programming in Education, PLI 2002, Pittsburgh PA, USA. (2002)
5. SICS, Swedish Institute of Computer Science: SICStus Prolog Manual, 3.10. (2003)
6. Colmerauer, A.: An introduction to Prolog III. Communications of the ACM **33** (1990) 69–90
7. Smullyan, R.M.: What is the Name of This Book? Prentice Hall, Englewood Cliffs, New Jersey (1978)
8. Hanák, D., Szeredi, T.: Finite domain constraint debugger. In: SICStus Manual ([5], Chapter 36). (2003)
9. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). In Podelski, A., ed.: Constraint Programming: Basics and Trends. Springer, Berlin, Heidelberg (1995) 293–316
10. Szeredi, P.: CLP resources (2003) `http://www.cs.bme.hu/~szeredi/clp.html`.
11. Harvey, W., Schimpf, J.: Eclipse sample code: Domino (2000) `http://www-icparc.doc.ic.ac.uk/eclipse/examples/domino.ecl.txt`.
12. Martin, P., Shmoys, D.B.: A new approach to computing optimal schedules for the job-shop scheduling problem. In: Proceedings of the 5th International IPCO Conference. (1996) 389–403
13. Simpson, S.: Nonogram solver (2003) `http://www.comp.lancs.ac.uk/computing/users/ss/nonogram/`.
14. Walsh, T.: Teaching by Toby Walsh: Nonogram solver (2000) `http://www-users.cs.york.ac.uk/~tw/Teaching/nonogram.html`.

A The Knights, Knaves and Normals Puzzle

This appendix describes a CLP(FD) program for solving a generic puzzle type, made popular by Raymond Smullyan, e.g. in [7].

The Puzzle. There is an island, on which there are three kinds of natives: *knights* always tell the truth, *knaves* always lie, and *normals* sometimes lie and sometimes tell the truth. Some local people make statements about themselves and we have to find out what kind of natives they are.

For example, in puzzle 39 from [7] three natives make the following statements:

A says: I am normal.
B says: That is true.
C says: I am not normal.

We know that A, B, and C are of different kinds. Determine who they are.

The Solution. In Fig. 14 we present a CLP(FD) program for solving puzzles of the above kind. Knights tell true statements, i.e. ones with the truth value 1, therefore knights are represented by the integer value 1. For analogous reasons, knaves are encoded as 0. Finally, a normal person is represented by the value 2. Statements are Prolog structures with the following syntax:

```
St ---> Ntv = Ntv | Ntv says St | St and St | St or St | not St
```

Here `St` denotes a statement, while `Ntv` is an integer or a constraint variable representing a native, i.e. one in the `0..2` range. The atoms "`says`", "`and`", "`or`", and "`not`" are declared operators. As syntactic sweeteners, we allow statements of form "`Ntv is Kind`", where `Kind` is one of the atoms `knight`, `knave`, or `normal`. This gets transformed to "`Ntv = Code`", where `Code` is the numeric encoding of `Kind`. To make the output more readable, we define an appropriate `portray` predicate.

The entry point of the program is the `says/2` predicate. The Prolog goal "`X says St`" expresses the constraint that native `X` says the statement `St`. With this interface the above concrete puzzle instance can be transformed into an "almost natural language" Prolog goal, as shown in Fig. 15.

B Performance Evaluation of the Domino Puzzle Solvers

In this Appendix we discuss the performance of various solutions of the domino puzzle, described in Sect. 5.

Test Cases. Table 3 shows the test sets used in the evaluation of the two domino solver variants. Altogether we have 63 test cases, grouped into four sets, according to their difficulty level. The difficulty seems to be most correlated to the number of solutions. The test cases can be downloaded from [10].

```
:- op(700, fy, not).              :- op(800, yfx, and).
:- op(900, yfx, or).              :- op(950, xfy, says).

% Native A can say sentence Sent.
A says Sent :- truth(A says Sent, 1).

% native(X): X is a native:
native(X) :- X in 0..2.

% truth(S, Value): The truth value of sentence S is Value.
truth(X = Y, V) :-
        native(X), native(Y), V #<=> (X #= Y).
truth(X says S, V) :-
        native(X), truth(S, V0),
        X #= 2 #\/           % Either X is normal or
        (V #<=> V0 #= X).    % the truth of what he says (V0) should be
                             % equal to him being a knight (X)
truth(S1 and S2, V) :-
        truth(S1, V1), truth(S2, V2), V #<=> V1 #/\ V2.
truth(S1 or S2, V) :-
        truth(S1, V1), truth(S2, V2), V #<=> V1 #\/ V2.
truth(not S, V) :-
        truth(S, V0), V #<=> #\ V0.
truth(X is Kind, V) :-
        code(Kind, Code), truth(X = Code, V).

% code(Kind, Code): Atom Kind is encoded by integer Code.
code(knave, 0).   code(knight, 1).   code(normal, 2).

portray(Code) :- code(Kind, Code), write(Kind).
```

Fig. 14. The CLP(FD) program for the knights, knaves, and normals puzzle

```
| ?- A says A is normal,
     B says A is normal,
     C says not C is normal,
     all_different([A,B,C]),
     labeling([], [A,B,C]).
A = knave, B = normal, C = knight ? ; no
```

Fig. 15. A run of the knights, knaves, and normals puzzle

Tuning Parameters. We evaluate the effect of various parameters on the performance of the presented solutions. The following is a list of tunable parameters and their possible settings. We will later refer to these using the phrases typeset in italics.

- labeling:
  - which kind of variables to label (only in the compass model): domino variables (DV) or compass variables (CV),
  - variable selection options: *leftmost*, *ff*, *ffc*

Table 3. Test sets for the Domino problem

Test set	Number of Tests	Best average time (sec)	Number of solutions (average)	Description
`base`	16	0.08	19	very basic tests for $d = 1..25$
`easy`	24	0.38	13	easy tests mostly of size $d = 15..25$
`diff`	22	41	110	difficult tests of size $d = 28, 30$
`hard`	1	332	1536	a very hard test of size $d = 28$

– shaving frequency (*freq.* =) *1, 2, 3*, ..., *once* (only before labeling), *none* (no shaving at all)
– values shaved (*shave* =)
  - (compass model): *[n,w]*, *[n]*
  - (border model): *[0]*, *[1]*, *[0,1]*
– implementation of the base constraint:
  - (compass model): implementing $X = c \Leftrightarrow Y = d$ through reification (*reif*), or by using an indexical for implication (*impl*), called twice, or by using an indexical for equivalence (*equiv*)
  - (border model): implementing the $\sum_n X_i = 1$ constraint using the library predicate `sum/3` (*libsum*), or using an FD-predicate for $n \leq 5$ and `sum/3` otherwise (*fdsum*)[10].

Performance. Table 4 presents performance results for some combinations of the above parameters, run with SICStus Prolog version 3.10.1 on a VIA C3 866 MHz processor (roughly equivalent to a 600 MHz Pentium III). For each test case all solutions were collected, with a CPU time limit of 1800 seconds (except for the hard test case, where the limit was 7200 seconds).

The table has three parts, presenting the results for the border model, for the compass model with DV labeling, and for the compass model with CV labeling. For each part, a full header line shows those parameter-settings which produced the best results. In subsequent lines the first column shows a (possibly empty) variation in the settings, while the subsequent columns show the performance results. Here two figures are given: the first is the total run-time in seconds, while the second one is the total number of backtracks, both for all test cases in the test-set. The $>$ symbol in front of the numbers indicates that there was a time-out while running at least one test case within the test-set. Note that backtracks during shaving are not counted.

Evaluation. Overall, the simpler border model seems to be faster than the compass model, except for the `hard` test case.

The border model involves Boolean variables only, so obviously the `ff` variable selection gives exactly the same search space as the `leftmost` one (as indicated by the same backtrack count). Intuitively, the `leftmost` strategy should

[10] The threshold of 5 seems to be optimal for SICStus Prolog. Additional measurements (not shown in the Table) give poorer results for thresholds of 4 and 6.

Table 4. Performance of the Domino solutions

Variation	`base`		`easy`		`diff`		`hard`	
border model, `leftmost`, freq. = 2, shave = [1], fdsum								
	1.56	1	9.26	8	910	1399	1373	2254
`ff`	1.53	1	9.18	8	910	1399	1351	2254
`ffc`	1.56	1	9.85	8	1792	2838	2181	3732
freq. = 1	1.63	1	9.59	3	1100	787	1642	1277
freq. = 3	1.53	1	9.28	20	931	2436	1370	3851
shave = [0]	1.29	2	11.04	103	1532	10719	2306	17300
shave = [0,1]	1.36	1	9.45	7	904	1324	1370	2150
libsum	2.75	1	13.85	8	1193	1399	1782	2254
freq. = once	1.41	1	10.93	1663				
freq. = none	2.68	818	49.73	21181				
compass model, DV labeling, `ff`, freq. = 3, shave = $[n, w]$, impl								
	3.17	1	18.57	19	2536	3597	332	477
`leftmost`	3.16	1	18.95	38	3389	8782	932	2547
`ffc`	3.19	1	17.34	17	>3790	>5374	2722	4095
freq. = 2	3.28	1	18.56	13	2516	2074	343	288
freq. = 4	3.15	1	19.06	41	2543	5720	353	727
shave = $[n]$	2.92	13	23.84	75	3971	11012	737	1820
reif	3.94	1	22.10	19	2670	3597	349	477
equiv	2.99	1	18.83	19	2691	3597	354	477
compass model, CV labeling, `ff`, freq. = 3, shave = $[n, w]$, impl								
	3.18	1	16.61	21	1684	2398	2570	3907
`leftmost`	3.18	1	16.59	21	1684	2398	2571	3907
`ffc`	3.18	1	19.28	25	>6606	>9840	>7200	>9343
shave = $[n]$	2.83	5	20.84	73	2600	6889	4609	12697

be faster, as its variable selection algorithm is simpler. In practice, both `ff` and `leftmost` settings give the same results (with a 2% variation, probably due to measurement errors).

For both models the best shaving frequency seems to be around 2-3. With a single shaving or no shaving at all, the performance is very poor.

In the border model, shaving the *[1]* value is bound to cause two constraints to fully instantiate all their variables, hence its performance is better than that of shaving *[0]*. On the other, hand shaving with both *[0,1]* gives a negligible reduction in the number of backtracks, and practically the same time as the variant shaving only the value *[1]*. In the compass model, shaving a single compass value (*[n]*), produces results inferior to those achieved when two non-opposite values (*[n,w]*) are shaved.

There are some open issues. For example, it would be interesting to find out those characteristics of the `hard` test case, which make the compass model more suitable for it. It also needs to be investigated why do we get very poor performance with the `ffc` variable selection strategy.

Reduce-To-The-Opt – A Specialized Search Algorithm for Contiguous Task Scheduling

Armin Wolf

Fraunhofer FIRST, Kekuléstraße 7, D-12489 Berlin, Germany
Armin.Wolf@first.fraunhofer.de

Abstract. A relevant class of real-world scheduling problems, e.g. for assembly lines, service centres but also for surgeries in operation rooms are characterised by *contiguous tasks*. These tasks are non-preemptive, non-overlapping, and especially without any time gaps. All tasks are performed without any breaks between any two tasks. Each task will start immediately after its direct predecessor, if there is any.
For these *contiguous task scheduling problems* a specialised search algorithm called Reduce-To-The-Opt is presented and its correctness is proved. Applied to trivial problems, where any linear ordering of the tasks solves the problem, this search algorithm also performs in linear time with respect to the number of tasks. For practical optimization problems, where the tasks have to be ordered optimally with respect to an objective function, runtime comparisons have shown that Reduce-To-The-Opt performs much better than a well-known heuristic-based search algorithm often used for the more general job-shop scheduling problems and even better than simple depth-first search because only the minimal value of each variable has to be selected instead of all possible values.

1 Introduction

Scheduling of tasks on exclusively available resources are in general NP-hard problems (cf. [3]), especially if they must be scheduled optimally with respect to an objective function like the sum of some setup costs. Nevertheless, constraint programming focuses on these problems, resulting in polynomial algorithms for pruning the search space (e.g. [3, 4, 6, 16]) and sophisticated search strategies (e.g. [13, 14]).

In this paper, we present an adaptation of the "Reduce-To-The-Max" algorithm presented in [14] for *contiguous task scheduling problems* like surgeries in an operation room (cf. Figure 1). These special scheduling problems are characterised by *contiguous task processing* without any time gaps. Furthermore, in this paper the correctness of this adapted algorithm called Reduce-To-The-Opt is proved.

The paper is organized as follows: After some preliminary definitions characterising the considered problems, the search algorithm is presented. Then the formal and informal descriptions of the algorithm are completed by the formal proof of its correctness. In addition, the algorithm is compared to other

Monday, 4th March 2002				
No.	Patient Code	Surgery	Start Time	Completion Time
1	80	inguinal hernia, left	8:00	8:45
2	78	fundoplication	8:45	10:15
3	79	fundoplication	10:15	12:45
4	77	appendectomy	12:45	13:15
5	81	inguinal hernia, right	13:15	14:00
6	82	inguinal hernia, right	14:00	14:30
7	83	umbilica hernia	14:30	15:30

Fig. 1. A typical surgery schedule

search strategies, especially with respect to a real-world application: the optimal scheduling of surgeries in an operation room with respect to some setup costs. The paper finishes with some concluding remarks.

2 Preliminaries

From an abstract point of view, a *contiguous task scheduling problem* is a constraint satisfaction problem (CSP) over finite domains: There are n tasks $t_1, \ldots,$ t_n having determined, positive integer-valued durations $d_1, \ldots, d_n$. The variable start times $s_1, \ldots, s_n$ of these tasks are constrained by non-negative, integer-valued finite domains, i.e. $s_i \in D_i$ for $i = 1, \ldots, n$, where

$$D_i = \{e_1^{(i)}, \ldots, e_{m_i}^{(i)}\} \ .$$

Supposing that $e_1^{(i)} < \cdots < e_{m_i}^{(i)}$ holds, $\min(D_i) := e_1^{(i)}$ is the obvious minimum of D_i.

Beyond other constraints O (e.g. representing objectives to be optimized) that must be respected, the tasks have to be scheduled non-overlapping and contiguous on the exclusively available resources (e.g. machines, employees, rooms, etc.). More formally, the CSP over finite domains that has to be solved consists of the constraints

$$\begin{array}{ll} O \wedge & \\ \bigwedge_{1 \le i \le n} & s_i \in D_i \wedge s_i \ge \mathsf{START} \wedge \\ \bigwedge_{1 \le i < j \le n} & (s_i + d_i \le s_j \vee s_j + d_j \le s_i) \wedge \\ \bigwedge_{1 \le i \le n} & s_i + d_i \le \mathsf{START} + \sum_{j=1}^{n} d_j \ , \end{array} \tag{1}$$

where the integer value START is the start time of the contiguous processing of the n tasks. Obviously, this means that we are looking for a permutation $\sigma : \{0, 1, \ldots, n\} \to \{0, 1, \ldots, n\}$ and a valuation $s_i = e^{(i)}$ with $e^{(i)} \in D_i$ such that

$$s_{\sigma(i)} = s_{\sigma(i-1)} + d_{\sigma(i-1)}$$

holds for $i = 1, \ldots, n$ under the assumption that there is either a real or a virtual task $t_{\sigma(0)}$ with $s_{\sigma(0)} + d_{\sigma(0)} = \mathsf{START}$. For instance, considering the schedule in Figure 1, the value of START represents 8:00 am.

In the following, we call this problem the n-dimensional *Contiguous Task Scheduling Problem* or CTSP for short. Furthermore, we call a CTSP *solvable* if such a permutation and such a valuation exists. In this case, the valuation is called a *solution* of this CTSP.

3 Reduce-To-The-Opt – A Solution of the CTSP

In the following we present the depth-first search algorithm Reduce-To-The-Opt that solves any CTSP correctly under some – very general and weak – assumptions on the behaviour of the underlying constraint solver: equivalence, domain pruning, and satisfaction testing of a propagation operator propagate:

```
Reduce-To-The-Opt($P$) $\equiv$
begin
(01) $P$ is a $n$-dimensional CTSP with start times $s_1, \ldots, s_n$;
(02) if $P =$ false then return;
(03) if $P = P' \wedge s_1 \in \{e^{(1)}\} \wedge \ldots \wedge s_n \in \{e^{(n)}\}$ then begin
(04)    print($s_1 = e^{(1)}$); ...; print($s_n = e^{(n)}$); println;
(05)    return;
(06) end
(07) else begin
(08)    $T := \{(s_{i_j}, D'_{i_j}) \mid P = P^{(i_j)} \wedge s_{i_j} \in D'_{i_j}, |D'_{i_j}| \geq 2\}$
(09)    $S := \{(s_q, D'_q) \mid \min(D'_q) \leq \min(D'_r)$ for all $(s_r, D'_r) \in T\}$;
(10)    Let $S = \{(s_{i_1}, D'_{i_1}) \ldots, (s_{i_k}, D'_{i_k})\}$;
(11)    for $p = 1, \ldots, k$ do begin
(12)       Reduce-To-The-Opt(propagate($P^{(i_p)} \wedge s_{i_p} \in \{\min(D'_{i_p})\}$));
(13)    end
(14)    return;
(15) end
end
```

Fig. 2. Reduce-To-The-Opt enumerates all solutions of a given CTSP

The algorithm works as follows: If the underlying constraint solver detects an inconsistency (Figure 2, line 02), the algorithm returns, signalling that the actual CTSP P is unsolvable. Otherwise, if all the tasks' start times are determined (Fig. 2, line 03, where P' is an according conjunction of constraints), a solution of the initially given CTSP is found and printed out. If neither an inconsistency is detected nor a solution is found, all the undetermined tasks' start times allowing a minimal value are gathered (Fig. 2, line 08–09, where $P^{(i_j)}$ are conjunctions of constraints according to the start times s_{i_j}). The corresponding tasks are the candidates to be scheduled next (Figure 2, line 10). The algorithm *fixes* all their minimal start times recursively (Fig. 2, line 12) until all valid start time combinations are printed (Fig. 2, line 04) out. In contrast to other search algorithms no other values than the minima are tried.

Theorem 1. *Let an n-dimensional CTSP C be given. Furthermore, assume that a so-called* propagation function $\mathsf{propagate}$ *exists which computes a fix-point while mapping CTSPs to equivalent CTSPs and restricting the domains of the start times (cf. [2]). In detail, supposing*

$$C = C' \wedge \bigwedge_{i=1,\dots,n} s_i \in D_i \ ,$$

where C' is an according conjunction of constraints, we assume that

- *the solutions of C and* $\mathsf{propagate}(C)$ *are the same,*
- $\mathsf{propagate}(C) = C' \wedge \bigwedge_{i=1,\dots,n} s_i \in D'_i$ *with $D'_i \subseteq D_i$ and* $\min(D'_i) \geq \mathsf{START}$*,*
- *if there is an $i \in \{1,\dots,n\}$ such that $D'_i = \{e^{(i)}\}$, i.e. $s_i = e^{(i)}$, then $D'_j \cap [e^{(i)}, e^{(i)} + d_i - 1] = \emptyset$ holds for each $j \in \{1,\dots,n\} \setminus \{i\}$,*
- *if C is solvable and $|D_1| = \cdots = |D_n| = 1$, then* $\mathsf{propagate}(C) = \mathsf{true} \wedge \bigwedge_{i=1,\dots,n} s_i \in D_i$*,*
- *if C is not solvable and $|D_1| = \cdots = |D_n| = 1$, then* $\mathsf{propagate}(C) = \mathsf{false}$.

Under these preconditions the algorithm Reduce-To-The-Opt(propagate(C)) *presented in Figure 2 enumerates* all *solutions of the CTSP C.*

Proof (by induction). **Induction base:** Given an one-dimensional CTSP

$$C = C' \wedge s_1 \in D_1 \ ,$$

where C' is a conjunction of constraints according to (1).

If C is not solvable, the algorithm (cf. Fig. 2) will return without printing any solution: either $|D_1| = 1$ and thus $P = \mathsf{propagate}(C) = \mathsf{false}$ in line 02 or the same holds after the recursive call of Reduce-To-The-Opt in line 12 because then $|D_1| = 1$.

If C is solvable, the algorithm (cf. Fig. 2) will not return in line 02 because $P := \mathsf{propagate}(C)$ returns a solvable CTSP

$$P = P' \wedge s_1 \in D'_1 \quad (D'_1 \subseteq D_1)$$

which is equivalent to C (P' is an according conjunction of constraints). Obviously, $s_1 = \mathsf{START}$ is the only solution of C and P. Thus, $\min(D'_1) \geq \mathsf{START}$ but $\min(D'_1) \not> \mathsf{START}$, i.e. $\min(D'_1) = \mathsf{START}$ holds and further it either holds that

1. $D'_1 = \{\min(D'_1)\} = \{\mathsf{START}\}$ (cf. line 03) or
2. $(s_1, D'_1) \in S$, where $S = \{(s_1, D'_1)\}$ (cf. line 09).

In the first case, the unique solution $s_1 = \mathsf{START}$ is printed out. In the second case, the recursive call of Reduce-To-The-Opt(propagate($P' \wedge s_1 \in \{\mathsf{START}\}$)) obviously prints out $s_1 = \mathsf{START}$. Thus in either case, Reduce-To-The-Opt terminates after printing out the unique solution.

Induction step: Given an $n+1$-dimensional CTSP

$$C = C' \wedge \bigwedge_{i=1,\ldots,n+1} s_i \in D_i \ ,$$

where C' is a conjunction of constraints according to (1).

If C is not solvable, then $P := \mathsf{propagate}(C)$ returns a CTSP

$$P = P' \wedge \bigwedge_{i=1,\ldots,n+1} s_i \in D'_i \quad (D'_i \subseteq D_i)$$

which is equivalent to C (P' is an according conjunction of constraints) and thus unsolvable, too. Obviously, any CTSP

$$E := P' \wedge \bigwedge_{i=1,\ldots,j-1,j+1,\ldots,n+1} s_i \in D'_i \wedge s_j \in \{\min(D'_j)\}$$

with $j \in \{1,\ldots,n+1\}$ is an unsolvable, n-dimensional CTSP. Thus by induction hypothesis, any recursive call of $\mathsf{Reduce\text{-}To\text{-}The\text{-}Opt}(\mathsf{propagate}(E))$ in line 12 will return without printing anything.

If C is solvable, the algorithm (cf. Fig. 2) will not return in line 02 because $P := \mathsf{propagate}(C)$ returns a solvable CTSP

$$P = P' \wedge \bigwedge_{i=1,\ldots,n+1} s_i \in D'_i \quad (D'_i \subseteq D_i)$$

which is equivalent to C (P' is an according conjunction of constraints). Obviously, there is at least one D'_p with $\min(D'_p) \leq \min(D'_j)$ for $j = 1,\ldots,n+1$. It either holds that

1. $D'_p = \{\min(D'_p)\}$ (cf. line 03) or
2. $(s_p, D'_p) \in S$ (cf. line 09).

In the first case, we have $s_p = \min(D'_p)$ and $\min(D'_p) < \min(D'_j)$ for all $j \neq p$ because $\min(D'_p) \leq \min(D'_j)$ for $j = 1,\ldots,n+1$ and moreover

$$D'_j \cap [\min(D'_p), \min(D'_p) + d_p - 1] = \emptyset$$

for any $j \neq p$, which results from the properties of the propagation function $\mathsf{propagate}$. Thus, for each solution of the CTSP C it holds that the task t_p is the first task in the contiguous task sequence. Otherwise, if the start $s(\sigma(1))$ of the first task is greater than $\min(D'_p) \geq \mathsf{START}$, then the completion time of the last task $s_{\sigma(n+1)} + d_{\sigma(n+1)}$ is greater than $\mathsf{START} + \sum_{j=1,\ldots,n+1} d_j$, which will contradict the premises. Thus, $s_{\sigma(1)} = \min(D'_p) = \mathsf{START}$ holds. Otherwise, if $\min(D'_p) > \mathsf{START}$ holds, then $s_{\sigma(n+1)} + d_{\sigma(n+1)} > \mathsf{START} + \sum_{j=1,\ldots,n+1} d_j$ will contradict the premises, too.

It follows directly that the initial call of $\mathsf{Reduce\text{-}To\text{-}The\text{-}Opt}$ is a call of $\mathsf{Reduce\text{-}To\text{-}The\text{-}Opt}$, where $P = \mathsf{propagate}(C)$ is a solvable n-dimensional CTSP (see below). Thus, by induction hypothesis the call of Reduce-To-The-Opt (propagate(C)) enumerates all solutions of this n-dimensional CTSP.

In the second case, let $(s_p, D'_p) \in S$, where $S = \{(s_{i_1}, D'_{i_1}), \ldots, (s_{i_k}, D'_{i_k})\}$ and $\min(D'_p) = \min(D'_{i_1}) = \cdots = \min(D'_{i_k})$ hold. Assuming, that $\min(D'_l) = \min(D'_p)$ holds for an arbitrary $l \in \{1, \ldots, n+1\} \setminus \{i_1, \ldots, i_k\}$. From the properties of propagate it follows that

$$D'_p \cap [\min(D'_l), \min(D'_l) + d_l - 1] = \emptyset$$

because D'_l must be singular, i.e. $|D'_l| = 1$. This contradicts $\min(D'_p) \in D'_p$, thus $\min(D'_l) > \min(D'_p)$ holds for all $l \in \{1, \ldots, n+1\} \setminus \{i_1, \ldots, i_k\}$ and all $p \in \{i_1, \ldots, i_k\}$.

Thus, there is at least one solution of the CTSP C such that the start time $s_{\sigma(1)}$ of the first task $t_{\sigma(1)}$ is in $\{s_{i_1}, \ldots, s_{i_k}\}$. Otherwise, it holds that $s_{\sigma(1)} > \min(D'_p) \geq \mathsf{START}$ and consequently it follows $s_{\sigma(n+1)} + d_{\sigma(n+1)} > \mathsf{START} + \sum_{j=1,\ldots,n+1} d_j$ contradicting the premises. Additionally, $s_{\sigma(1)} = \min(D'_p) = \mathsf{START}$ holds, otherwise the premises are contradicted again, i.e. $s_{\sigma(n+1)} + d_{\sigma(n+1)} > \mathsf{START} + \sum_{j=1,\ldots,n+1} d_j$ will follow. Thus, all tasks that are first in any solution are in $\{t_{i_1}, \ldots, t_{i_k}\}$. Furthermore, the n-dimensional CTSPs $P^{(p)} \wedge s_p \in \{\min(D'_p)\}$ as well as the equivalent CTSP propagate($P^{(p)} \wedge s_p \in \{\min(D'_p)\}$)) is solvable for any task $t_p \in \{t_{i_1}, \ldots, t_{i_k}\}$, if t_p is the first task in at least one solution. Thus by induction hypothesis, the recursive calls of Reduce-To-The-Opt(propagate($P^{(p)} \wedge s_p \in \{\min(D'_p)\}$) in line 12 will print out all solutions of these n-dimensional CTSPs, where t_p is the first task. Thus, during the iteration over all tasks that might be first, all solutions of the original CTSP are printed out.

In all cases, the n-dimensional CTSPs are determined by

$$\begin{array}{ll} O \wedge & \\ \bigwedge_{1 \leq i \leq n+1, i \neq p} & s_i \in D'_i \wedge s_i \geq \mathsf{START} + d_p \wedge \\ \bigwedge_{1 \leq i < j \leq n+1, i \neq p, j \neq p} & (s_i + d_i \leq s_j \vee s_j + d_j \leq s_i) \wedge \\ \bigwedge_{1 \leq i \leq n+1, i \neq p} & s_i + d_i \leq \mathsf{START} + d_p + \sum_{j=1, j \neq p}^{n+1} d_j \end{array} ,$$

where $\mathsf{START} + d_p$ is the start time of the contiguous processing of the remaining n tasks after the first task t_p. □

4 Related Work

Comparing the presented search algorithm in Figure 2 with other search algorithms for task scheduling problems (e.g. [3] as well as [9] for a general schema, or [7] for a specific variant) two essential differences are remarkable:

1. During the search, orderings between unordered tasks are established, which requires the detection of unordered pairs of tasks and the propagation of additional constraints.
2. Additional heuristics are used to select the next unordered pair and to determine which task should be before the other.

Thus, with respect to the first item, our algorithm is much simpler because we do not care about any tasks orderings. Furthermore, if $O = \mathsf{true}$ in (1), the Reduce-To-The-Opt search algorithm will find a first solution of the n-dimensional CTSP in linear time, i.e. the runtime complexity is $O(n)$, because the first candidate in all loops (line 11–13 in Fig. 2) during any recurive calls will lead to a solution.

With respect to the second item, it is also possible to add some heuristics to select the start time to be considered next (cf. Fig. 2, lines 10–11). For instance, the start time of "...the activity with the smallest latest start time is chosen when two or several activities share the same release date" [3] will be a appropriate candidate.

However, the crucial difference between Reduce-To-The-Opt and other search algorithms available in CP systems like CHIP [1], ILOG Solver/Scheduler [9], mOZart [15], SICStus Prolog [11] is the fixation of the variables' values: Of course, the selection of a variable is based on the well-known heuristic "select a variable with the smallest minimal number" (cf. [9], Page 72) and the variable's value is selected to be this minimal number. But in contrast to all these algorithms, the value selection is not changed during the whole search process, i.e. to be the second smallest number, etc. (cf. [15], Sections 2.6–2.8). In Reduce-To-The-Opt the variable's value is fixed to be its minimal value, which results in general, i.e. for arbitrary CSPs, into an incomplete search loosing some or all solutions but is complete and correct for CTSPs (see Theorem 1).

As well as any other search algorithm for task scheduling our algorithm is non-polynomial in the worst case because the solved problems are in general NP-hard. Especially, if we consider optimization problems, where we are interested in a *best* solution with respect to an objective function and not only in any solution. In this case, there might be $n!$ trials necessary to find an optimal solution of an n-dimensional CTSP.

In the following, we describe a practial optimization problem where we compare Reduce-To-The-Opt with a more general job-shop scheduling algorithm.

5 Runtime Comparison on Surgery Scheduling Problems

The practical contiguous tasks scheduling problems that we solved are surgery scheduling problems. Here, different surgeries have to scheduled contiguously such that some sequence-dependent setup costs (non-negative integer values) are minimized. These sequence-dependent setup costs are the costs for the reconfiguration of the used operation tables. In the considered case, there are two tables which are used alternately: During a surgery, when one table is in use, the other table might be reconfigured and the next patient is prepared. After the finish of the surgery the tables are exchanged. Then, the used table becomes available for any reconfiguration for the following surgery. Thus, if there are only two configurations necessary, e.g. left/right arm, any solution where these tables are used alternately, e.g. left, right, left, $\ldots$, or right, left, right, $\ldots$, is optimal with respect to the setup costs: both tables must be configured only once, i.e.

Thursday, 28th March 2002					
No.	Patient Code	Surgery	Start Time	Completion Time	Configuration
1	229	inguinal hernia, left	8:00	8:45	left arm
2	231	inguinal hernia, right	8:45	9:30	right arm
3	227	inguinal hernia, left	9:30	10:15	left arm
4	232	umbilica hernia	10:15	11:15	right arm
5	228	inguinal hernia, left	11:15	12:00	left arm
6	233	lapraroscopy	12:00	12:45	right arm
7	230	inguinal hernia, left	12:45	13:30	left arm

Fig. 3. An optimal surgery schedule with respect to configuration costs

before their first use (see Fig. 3). In this practical application some more temporal constraints and optimization criteria have to be considered, however they are confidential and thus not published.

A more abstract point view gives some evidence that this optimization problem is in general an NP-hard problem: it corresponds with some kind of *asymmetric Travelling Salesman Problem* (TSP), which is known to be in general a NP-hard problem.

Recapturing TSPs, we are interested in closed paths in a directed, weighted graph where the first and the last nodes are the same and all other nodes are visited exactly once such that the sum of the edges' weights is minimal. In our case, the nodes are the surgeries and additionally there is an "initial" node. There are directed edges betweeen any pair of different nodes. The directed edges between two surgeries are weighted with the corresponding setup costs. Any edge from the inital node to a surgery is weighted with the inital setup cost of this surgery. Any edge from a surgery to the inital node has zero weight.

Given $2k(+1)$ surgeries, the optimal surgery scheduling problem can be seen in this abstract model as finding a path with a minimal sum of weights starting at the initial node, returning to the inital node after visiting $k(+1)$ different surgeries and returning again to the inital node after visisting the remaining k surgeries. Interleaving the $k(+1)$ and the k surgeries on both sub-paths results in an optimal surgery schedule with respect to the setup costs. It should be noted that it is a priori unknown which surgery belongs to which path.

We use a dichotomizing extension of the Reduce-To-The-Opt search algorithm to solve this NP-hard problem, i.e. finding optimal schedules of the introduced surgery scheduling problems: The objective function, here the sum of setup costs[1], is formulated as an additional constraint (cf. O in (1)), where the sum is equated to an additional constraint variable *obj*. Furthermore, a lower and an upper bound (*lwb*/*upb*) of the value of this constraint variable are maintained. Initially, the lower bound is zero and the upper bound is either trivially chosen or the cost of any first found solution. After finding a solution, search is cancelled and the constraint variable *obj* is constrained to be less than or equal to the sum of both bounds divided by two:

$$obj \leq \lfloor (lwb + upb) \div 2 \rfloor$$

[1] and some other costs

The search is reactivated to find a better solution. If there is a better solution the upper bound becomes the cost of this solution. Otherwise, if there is no better solution, the posted inequality is rejected and the lower bound becomes the sum of both bounds divided by two plus one. In both cases, dichotomization continues until the lower bound is equal to the upper bound. Then, the recently found solution is optimal.

The application of this algorithm to some real-world surgery scheduling problems have shown that it behaves very efficient, especially compared to dichotomizing extensions of other search algorithms:

– to "standard labeling" as in the ECL^{i}PS^e system [10]; i.e. simple depth-first search, where the tasks are considered in the order of their occurence and where their possible start times are tried in ascending order,
– to heuristic-based job-shop scheduling as presented in [3] and further extended by "shaving" as proposed in [5, 12].

The results are very satisfying in this application area; it is possible to schedule a daily used operation room optimally for the time period of a month with respect to the setup costs in less than a second on a modern Pentium III PC. The more general job-shop scheduling algorithm mentioned before requires about 30 seconds for the same optimization problem. "Standard labelling", which potentially tries all values of a variable, is unable to find the optimal schedules within a 30 minutes processing time.

6 Conclusion and Future Work

In this paper we presented the novel search algorithm Reduce-To-The-Opt for contiguous task scheduling problems based on formerly presented ideas realized in the "Reduce-To-The-Max" search algorithm. The main contribution of this paper is the formal proof of its correctness. The proof is based on some rather weak assumptions to the underlying constraint solver. From our knowledge, these assumptions are valid for all commonly available constraint systems supporting constraint-based scheduling like CHIP, ECLiPSe, ILOG Solver/Scheduler, mOZart, SICStus Prolog, Prolog III as well as for our own constraint solver `firstcs` [8].

Furthermore, the presented search algorithm is applied to solve some real-world scheduling problems, e.g. the optimal scheduling of surgeries in operation rooms with respect to setup costs. This application shows that our specialised algorithm outperforms other search algorithms by orders of magnitudes.

Future work will examine whether an adoption of some algorithms for *asymmetric* TSPs is possible such that they will solve the surgery scheduling problems, which are modelled in Section 5 as specialized TSPs. If so, further runtime comparisons will be performed.

References

1. *CHIP System Documentation – Volume 1 – CHIP++ Reference Manual.* Parc Club Orsay Université, 4, rue Jean Rostand, 91893 ORSAY Cedex, France, version 5.4, August 2001.

2. Krzysztof R. Apt. From chaotic iteration to constraint propagation. In *Proceedings of 24th International Colloquium on Automata, Languages and Programming (ICALP '97)*, number 1256 in Lecture Notes in Computer Science, pages 36–55. Springer-Verlag, 1997.
3. Philippe Baptiste, Claude le Pape, and Wim Nuijten. *Constraint-Based Scheduling.* Number 39 in International Series in Operations Research & Management Science. Kluwer Academic Publishers, 2001.
4. Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming - CP2001*, number 2239 in Lecture Notes in Computer Science, pages 377–391. Springer Verlag, 2001.
5. J. Carlier and E. Pinson. Adjustments of heads and tails for the job-shop problem. *European Journal of Operational Research*, (78):146–161, 1994.
6. Yves Caseau and François Laburthe. Improved CLP scheduling with task intervals. In Pascal van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming, ICLP'94*, pages 369–383. MIT Press, 1994.
7. Yves Colombani. Constraint programming: an efficient and practical approach to solving the job-shop problem. In Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*, volume 1118, pages 149–163. Springer Verlag, August 1996.
8. Matthias Hoche, Henry Müller, Hans Schlenker, and Armin Wolf. firstcs - A Pure Java Constraint Programming Engine. In Michael Hanus, Petra Hofstedt, and Armin Wolf, editors, *2nd International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL'03*, 29th September 2003. Online available at `uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf`.
9. ILOG SA, 9, rue de Verdun, BP 85, 94253 Gentilly Cedex, France. *ILOG Scheduler – User's Manual*, version 5.0, July 2000.
10. International Computers Limited and Imperical College London. *ECL^iPS^e Constraint Library Manual*, release 5.7, December 2003.
11. Intelligent Systems Laboratory. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, PO Box 1263, SE-16429 Kista, Sweden, release 3.11.0, October 2003.
12. P. D. Martin and D. B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings of the 5th Conference on Integer Programming and Combinatorical Optimization*, 1996.
13. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).
14. Hans Schlenker. Reduce-To-The-Max: Ein schneller Algorithmus für Multi-Ressourcen-Probleme. In Francois Bry, Ulrich Geske, and Dietmar Seipel, editors, *14. Workshop Logische Programmierung*, number 90 in GMD Report, pages 55–64, 26th–28th January 2000.
15. Christian Schulte and Gert Smolka. *Finite Domain Constraint Programming in OZ – A Tutorial.* The Mozart Programming System, version 1.2.5, 25th January 2003.
16. Armin Wolf. Pruning while sweeping over task intervals. In Francesca Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming – CP 2003*, number 2833 in Lecture Notes in Computer Science, pages 739–753, Kinsale, County Cork, Ireland, 30th September – 3rd October 2003. Springer Verlag.

A New Approach to Modeling and Solving Minimal Perturbation Problems

Roman Barták[1], Tomáš Müller[1], and Hana Rudová[2]

[1] Charles University, Faculty of Mathematics and Physics
Malostranské nám. 2/25, Prague, Czech Republic
{bartak,muller}@ktiml.mff.cuni.cz
[2] Masaryk University, Faculty of Informatics
Botanická 68a, Brno, Czech Republic
hanka@fi.muni.cz

Abstract. Formulation of many real-life problems evolves when the problem is being solved. For example, a change in the environment might appear after the initial problem specification and this change must be reflected in the solution. Such changes complicate usage of a traditionally static constraint satisfaction technology that requires the problem to be fully specified before the solving process starts. In this paper, we propose a new formal description of changes in the problem formulation called a minimal perturbation problem. This description focuses on the modification of the solution after a change in the problem specification. We also describe a new branch-and-bound like algorithm for solving such type of problems.

Introduction

Many real-life problems can be naturally stated as constraint satisfaction problems (CSP). In practice, the problem formulation is not static but it evolves in time [5,6,12,13,17]. In fact, it evolves even during solving the problem. Thus, to further spread up applicability of the constraint satisfaction technology in real-life applications, it is necessary to cover the dynamics of the real world. In particular, it is necessary to handle changes in the problem specification during the solving process.

The problem changes may result from the changes in the environment like broken machines, delayed flights, and other unexpected events. The users may also initiate some other changes that might specify new properties of the problem based on a (partial) solution found so far. The goal is to find a better solution for the users. Naturally, the problem solving process should continue as smoothly as possible after any change in the problem formulation. In particular, the solution of the altered problem should not differ much from the solution found for the original problem.

There are several reasons to keep the new solution as close as possible to the existing solution. For example, if the solution has already been published like the assignment of gates to flights then it would not be convenient to change it frequently because it would confuse passengers. Moreover, the changes to the already published solution might force other changes because the originally satisfied wishes of the users may be violated which raise an avalanche reaction.

Our work is motivated by a large scale timetabling problem at Purdue University. Once timetables are published there, they require many changes based on the additional user input. These changes should be reflected in the problem solution with a minimal impact on any previously generated solution. Thus, the primary focus of our work is to provide a support for making such changes to the generated timetable. The basic requirement is to keep the solution as close as possible to the published solution of the initial problem provided that the new solution is a solution of the altered problem. In terms of constraint satisfaction, it means that the minimal number of variable assignments is changed after the problem modification.

The paper studies the above type of dynamic problems called a minimal perturbation problem (MPP). The basic task is to find a solution of the altered problem in such a way that this new solution does not differ much from the solution of the original problem. In addition to the formal definition of the minimal perturbation problem, we also propose a branch-and-bound like algorithm to solve such problems. This algorithm can provide an approximate solution for over-constrained and hard-to-solve problems too.

The paper is organized as follows. We first give more details about the course timetabling problem that makes the main motivation of our research. Then, we survey the existing approaches to the minimal perturbation problem and to handling dynamic changes in CSP. The main part of the paper is dedicated to a formalization of the minimal perturbation problem and to a description of the solving algorithm for such a problem. We conclude with an experimental evaluation of the algorithm using random placement problems.

Motivation: A Course Timetabling Problem

The primary intent behind our work on minimal perturbation problems lies in the need to solve such a problem in the context of a real timetabling application for Purdue University (USA). The timetabling problem at Purdue University consists of allocating approximately 750 courses into 41 large lecture rooms with capacities up to 474 students. The courses are taught several times a week resulting in about 1,600 meetings to be scheduled. The space covered by all the meetings fills approximately 85% of the total available space.

There are special meeting patterns defined for each course that restrict possible time and location placement. For example, the valid combinations of days per course are given and all meetings of the same course must be taught at the same classroom and at the same time of the day. The classroom allocation must also respect the instructional requirements and the preferences of the faculty. Moreover, the instructors may have specific time requirements and preferences for each course.

The students select the courses that they wish to attend. The task is to schedule all the courses respecting given constraints and preferences while minimizing the number of potential student course conflicts for each of almost 29,000 students. The conflict appears when a student wishes to attend different courses that overlap in time.

The construction of a solution for the above problem was described in [15]. The main ideas behind the constraint model are as follows. Each meeting of a course is described using two domain variables: a time variable (starting time of the meeting in a week) and a classroom variable. The hard constraints ensure that two meetings will not be taught at the same classroom at once. The constraints also remove the values prohibited by the meeting patterns and by the requirements of instructors and faculties. The preferences on time and classroom placement together with the student course conflicts are modeled using soft constraints. The cost of a soft constraint ensuring that two courses with common students will not overlap is equal to the number of common students. The preferences of instructors and faculties are expressed using soft unary constraints.

The variable assignment is constructed using a new limited assignment number (LAN) search algorithm [18]. The LAN search algorithm is an incomplete iterative search algorithm where the standard backtracking is limited to an incomplete search of the linear complexity. The aim is to generate a partial solution with the maximal number of assigned variables. The algorithm runs in iterations where each iteration step explores some sub-tree of the search tree. The linear complexity is achieved by considering a limit on the number of assignments for each variable tried during each iteration step. The result of one iteration step is a partial assignment that is used as a guide in the next iterations. Special value and variable ordering heuristics have been proposed for this purpose.

It may still happen that the set of hard constraints is over-constrained. The user input can be used to resolve this problem by relaxing some constraints. Then, the LAN search algorithm can continue in the subsequent iterations with the problem definition changed. This approach is similar to the problem considered in this paper – we have a partial solution and we want to construct a new solution under the redefinition of the problem. However, the approach of the LAN search algorithm does not attempt to minimize the number of changes in the subsequent solutions. It rather maximizes the number of assigned variables. A solver for minimal perturbation problems should do both tasks.

The solver of the above timetabling problem was implemented using the `clpfd` library of SICStus Prolog [2] with the help of our soft constraints solver [16]. The initial implementation started with built-in backtracking of SICStus Prolog. However, the experiments with standard backtracking did not lead to a solution after 10 hours of CPU time because too many failed computations were repeated exploring the parts of the search tree with no solution. The LAN search algorithm was able to substantially improve on the initial partial solution. Starting from 33 courses, only one course remained unassigned after eight iterations of the LAN search algorithm. Assignment of this course was successfully completed with the help of the user.

The assignment generated by the LAN search algorithm introduces an initial solution of the timetabling problem. Once this solution is published, it requires many changes based on the additional input. These changes should be involved in the problem solution with a minimal impact on any previously generated solution. Note also, that it is not possible to limit the changes in the problem definition. Faculties and instructors may come with completely new courses to be scheduled and the requirements towards the original courses may change substantially. Moreover, some requirements or classes may be canceled which can help to find a more acceptable solution. On the other hand, the new requirements usually make the problem harder since we are constrained in the number of allowed changes with respect to the original solution. The original problem, its solution, and the set of requested changes introduce the input for the minimal perturbation problem we are going to solve.

The LAN search algorithm represents the very first step in the direction towards solving the minimal perturbation problem because the partial solution generated in each iteration step is derived from the previous solution. As our experiments showed, the distance of these solutions is mostly acceptable if a small number of changes is introduced to the problem. However, the distance significantly enlarges when the number of problem changes increases. Other experiments also showed that a more informed heuristics may improve the quality of the solution substantially. Thus, the LAN Search seems to be a good base for solving the minimal perturbation problem.

Related Works

Dynamic problems appear frequently in real-life planning and scheduling applications where the task is to "minimally reconfigure schedules in response to a changing environment" [7]. Therefore, handling dynamic changes of the problem formulation was studied in the constraint community for some time. Dechter and Dechter [4] proposed a notion of *Dynamic Constraint Satisfaction Problem* (DCSP) that is a sequence of CSPs, where every CSP is a result of changes in the preceding one. A difference between two consecutive CSPs is expressed by a set C_{add} of added constraints and a set C_{del} of constraints deleted from the problem.

There exist many criteria for evaluating the performance and quality of algorithms solving DCSP, like efficiency in finding a new solution or solution stability in the face of problem alternation. A minimal perturbation problem can occur here as a criterion of similarity or consistency of the solution – the smallest number of values should be different between any new and original solution.

Several algorithms have been proposed to search for a minimum change in the solution. For example, Ran, Roos, and Herik [14] proposed an algorithm for finding a near-minimal change solution that is a solution with the minimal or near minimal number of modified value assignments. Their algorithm is looking for a new solution by changing the assignment of one, two, tree variables, and so on until a feasible assignment is found – an approach similar to Limited Discrepancy Search [10].

A *minimal perturbation problem* was described formally by El Sakkout, Richards, and Wallace in [6] as a 5-tuple $(\Theta, \alpha, C_{add}, C_{del}, \delta)$, where Θ is a CSP, α is a solution to Θ, C_{add} and C_{del} are constraint removal and addition sets, and δ is a function that measures the distance between two complete assignments. Their solving method combining linear and constraint programming is looking for a complete assignment β that minimizes $\delta(\alpha,\beta)$ and that is a solution of the new problem arising from Θ by adding the constraints from C_{add} and deleting the constraints from C_{del}.

It is interesting to see that there is a lack of works concerned by search for a minimum change [14,7]. A commented bibliography from [17] refers only to four papers including an earlier version of this paper.

Looking to existing methods to handle dynamic problems, there are two main differences in our approach. First, we allow arbitrary changes of the problem formulation including addition and deletion of the variables. Note that variable deletion can be modeled in DCSP as a deletion of all the constraints containing the variable. Variable addition cannot be modeled directly in DCSP. Second, we are working with incomplete assignments to solve over-constrained and hard-to-solve problems. Other methods usually relax constraints in advance to solve such problems.

A Formal Model

In this section, we present a new formal model of the minimal perturbation problem (MPP) that is applicable to over-constrained problems as well as to problems where finding a complete solution is hard. Recall that the idea of MPP is to define a solution of the altered problem in such a way that this solution is as close as possible to the solution of the original problem. We first survey the standard definitions of CSP and we introduce a new notion of a maximal consistent assignment. This new notion will help us to describe formally an approximate solution of the constraint satisfaction problem. In the second part, we define a minimal perturbation problem and a solution of the minimal perturbation problem.

Preliminaries

A *constraint satisfaction problem (CSP)* is a triple $\Theta = (V,D,C)$, where

- $V = \{v_1, v_2,\ldots, v_n\}$ is a finite set of variables,
- $D = \{D_1, D_2,\ldots, D_n\}$ is a set of domains, where D_i is a set of possible values for the variable v_i,
- $C = \{c_1, c_2,\ldots, c_m\}$ is a finite set of constraints restricting the values that the variables can simultaneously take.

A *solution* to the constraint satisfaction problem Θ is a complete assignment of the variables from V that satisfies all the constraints.

For many constraint satisfaction problems it is hard or even impossible to find a solution in the above sense. For example, it does not exist any complete assignment satisfying all the constraints in over-constrained problems. Therefore other definitions of the problem solution, for example Partial Constraint Satisfaction [8], were introduced.

In this paper, we propose a new view of the problem solution based on a new notion of a maximal consistent assignment. This approach is strongly motivated by the course timetabling problem but we believe that it is generally applicable. The basic idea is to assign as many variables as possible while keeping the problem consistent. Then, the user may relax some constraints in the problem – typically some of the constraints among the non-assigned variables that cause conflicts – so that after this change the partial assignment can be extended to other variables.

Formally, let Θ be a CSP and C be a consistency technique, for example arc consistency. We say that the *constraint satisfaction problem* is *C-consistent* if the consistency technique C deduces no conflict. For example, for arc consistency the conflict is indicated by emptying a domain of some variable. We denote $C(\Theta)$ the result of the consistency test which could be either *true*, if the problem Θ is C-consistent, or *false* otherwise. Let Θ be a CSP and σ be an assignment of some variables from Θ. Then, we denote $\Theta\sigma$ an application of the assignment σ to the problem Θ. $\Theta\sigma$ is a problem derived from Θ such that the domains of the variables from σ are reduced to a value defined by σ. Finally, we say that an *assignment* σ is *C-*

defined by σ. Finally, we say that an *assignment σ* is *C-consistent* with respect to some consistency technique C if and only if C($\Theta\sigma$) is true. Note that a complete C-consistent assignment is a solution of the problem provided that the consistency technique C is able to check satisfaction of the constraints when the values of the variables are known. Backtracking-based solving techniques are typically extending a partial consistent assignment towards a complete consistent assignment where all the variables are assigned.

As we already mentioned, for some problems there does not exist any complete consistent assignment – these problems are called *over-constrained.* In such a case, we propose to look for a solution defined using the notion of a maximal consistent assignment. We say that a *C-consistent assignment is maximal* for a given CSP if there is no C-consistent assignment with a larger number of assigned variables. We can also define a weaker notion of a so called *locally maximal C-consistent assignment.* A locally maximal C-consistent assignment is a C-consistent assignment that cannot be extended to another variable.

Notice the difference between the above two notions. The maximal C-consistent assignment has a global meaning because it is defined using the cardinality of the assignment that is the number of assigned variables is maximized there. The locally maximal C-consistent assignment is defined using a subset relation – it is not possible to extend the locally maximal C-consistent assignment by assigning any additional variable without getting inconsistency. Visibly, the maximal C-consistent assignment is the largest (using cardinality) locally maximal C-consistent assignment.

Example (maximal arc-consistent assignments):

Let V = {a,b,c,d,e} be a set of variables with domains D = {D_a={1,2}, D_b={1,2,3}, D_c={2,3}, D_d={2,3}, D_e={2,3}} and C = {a≠b, b≠c, c≠d, c≠e, d≠e} be a set of constraints. Assume that we use arc consistency as the technique for checking consistency of CSP Θ = (V,D,C). Then:

- σ = {a/1} is a locally maximal arc-consistent assignment for Θ which is not a maximal arc-consistent assignment ($|\sigma|=1$),
- γ = {a/2, b/1} is a maximal arc-consistent assignment for Θ ($|\gamma|=2$).

If a constraint satisfaction problem has a solution then any maximal C-consistent assignment is the solution provided that the consistency technique C is able to check satisfaction of the constraints when the values of the variables are known. If a constraint satisfaction problem has no solution – it is an over-constrained problem – then some maximal C-consistent assignment still exists. We propose to define the *solution of the (over-constrained) problem* as a maximal C-consistent assignment.

There is a strong real-life motivation for the above view of the solution for over-constrained problems. In scheduling and timetabling applications [13,15], the maximal C-consistent assignment usually corresponds to allocation of the largest number of activities to resources. Such an assignment is more informative than the answer "no" that indicates non-existence of a complete assignment of variables satisfying the constraints. Moreover, if the problem solution is defined as a maximal C-consistent assignment then some solution always exists. In particular, it is not necessary to know in advance that the problem is over-constrained. So, a maximal C-consistent assignment is as a generalization of the traditional solution. It covers a solution of CSP as well as a solution of over-constrained CSP.

Because the maximal C-consistent assignment is the largest among the locally maximal C-consistent assignments, a locally maximal C-consistent assignment can be seen as an *approximate solution*. The largest the assignment is, the better approximate solution we have. The constraint solver may return an approximate solution when the problem is hard-to-solve. In particular, the solver may return the largest locally maximal C-consistent that is possible to compute using given resources (e.g., time).

Notice finally that the solution is parameterized by a consistency technique C which gives users the flexibility to define the desired features of the solution. The consistency technique may check only validity of the constraints between the assigned variables. Then we get the assignment with the largest number of instantiated variables. This assignment cannot be extended to another variable without getting a constraint violation. If a stronger consistency technique is used, for example arc consistency, we may expect a shorter assignment as the solution. However, this assignment can be extended to another variable without getting a constraint violation. Instead, we get a failure of the arc-consistency test. In some sense, a solution defined using a stronger consistency level gives the user some advance for future extensions of the assignment.

A Minimal Perturbation Problem

Now we can formally define a *minimal perturbation problem* (MPP) as a triple $\Pi = (\Theta, \alpha, \delta)$, where:

- Θ is a CSP,
- α is a possibly partial assignment for Θ called an *initial assignment*,
- δ is a *distance function* defining a distance between any two assignments.

A *solution to the minimal perturbation problem* $\Pi = (\Theta, \alpha, \delta)$ is a solution β for Θ such that $\delta(\alpha,\beta)$ is minimal. Recall, that we define the solution as a maximal C-consistent assignment for some consistency technique C. The idea behind the solution of MPP is apparent – the task is to find the largest possible assignment of variables for the problem Θ in such a way that it differs minimally from the initial assignment.

Recall that the minimal perturbation problem should formalize handling changes in the problem formulation. So, one may ask where the original problem is in the above definition. Because the new solution is compared only to the solution of the original problem, it is not necessary to include neither the original problem nor the description of the problem change in the definition of MPP. This gives us the freedom to change the problem arbitrarily, in particular to add and remove variables and constraints and to change variables' domains. The solution of the original problem can be mapped to an initial assignment in Π in the following way. Assume that an assignment σ is a solution of the original problem. Then the initial assignment α in the definition of MPP is defined in the following way:

$$\alpha = \{v/h \mid v/h \in \sigma \ \& \ v \in \Theta\}^1.$$

[1] For simplicity reasons we write $v \in \Theta$ which actually means $v \in V$, where $\Theta = (V,D,C)$.

Note that α and σ are not necessarily identical because some variables may be removed from the problem. Thus, $\alpha \subseteq \sigma$ holds.

The distance function δ in the definition of MPP is specified by the user. For purposes of our timetabling problem, we use a specific distance function describing the number of differences between two assignments. Let σ and γ be two assignments for Θ. Then we define $W(\sigma,\gamma)$ as a set of variables v such that the assignment of v in σ is different from the assignment of v in γ:

$$W(\sigma,\gamma) = \{v \in \Theta \mid v/h \in \sigma \ \&\ v/h' \in \gamma \ \&\ h \neq h'\}.$$

We call $W(\sigma,\gamma)$ a *distance set* for σ and γ and the elements of the set are called *perturbations*. The distance function is then defined in the following way:

$$\delta(\sigma,\gamma) = |W(\sigma,\gamma)|.$$

If a metric is defined on the variables' domains then it is possible to specify other distance functions, for example:

$$\delta(\sigma,\gamma) = \max_v \{dist_v\ (h,h') \mid v/h \in \sigma \ \&\ v/h' \in \gamma\}, \text{ or}$$
$$\delta(\sigma,\gamma) = \textstyle\sum_v \{dist_v\ (h,h') \mid v/h \in \sigma \ \&\ v/h' \in \gamma\}, \text{ or}$$
$$\delta(\sigma,\gamma) = (\textstyle\sum_v \{dist_v^2(h,h') \mid v/h \in \sigma \ \&\ v/h' \in \gamma\})^{1/2},$$

where $dist_v$ is a distance function (metric) on the domain of the variable v.

Notice that the above formulation of MPP generalizes the formulation from [6] by working with partial assignments rather than with complete assignments and by allowing arbitrary changes to the problem. Also, we reformulated the definition from [1] to be easier and more general while preserving the original meaning that is minimizing the number of changes in the solution after a problem change

Example:

Let $\alpha=\{b/3\}$ be an initial assignment for a CSP Θ with variables {b,c,d}, domains $\{D_b=\{1,3\}, D_c=\{1,2,3\}, D_d=\{2,3\}\}$, and constraints $\{b\neq c, c\neq d, d\neq b\}$. Then the problem Θ has the following solutions (maximal arc-consistent assignments):

- $\beta_1 = \{b/1,c/2,d/3\}$ $(W(\alpha,\beta_1) = \{b\})$,
- $\beta_2 = \{b/1,c/3,d/2\}$ $(W(\alpha,\beta_2) = \{b\})$,
- $\beta_3 = \{b/3,c/1,d/2\}$ $(W(\alpha,\beta_3) = \{\})$,

but only the solution β_3 is a solution of MPP $\Pi = (\Theta, \alpha, |W|)$.

MPP Solver

A minimal perturbation problem is a type of optimization problem so it is natural to use optimization technology to solve it. In particular, we have modified a branch-and-bound algorithm for this purpose.

There is one specialty of MPP going beyond conventional objective functions, in particular the maximization of the number of assigned variables. Handling incomplete assignments is important for solving over-constrained problems. It also helps to produce an approximate solution for hard-to-solve problems. The proposed algorithm explores locally maximal consistent assignments and it keeps the largest assignment with a minimal distance from the initial assignment (row 4). We have developed a concept of variable locking to extend any partial consistent assignment to a locally maximal consistent assignment.

The algorithm is also requested to run in an interactive environment that is the solution must be produced in a reasonable time. Often a complete search space is too large. Therefore, we combine the optimization branch-and-bound algorithm with principles of the LAN search algorithm [18]. In particular, we propose a concept of variable expiration that decreases the size of the search space. Consequently, the algorithm finds an approximate solution as defined in the previous section.

```
    label(Variables,LockedVariables)
1       if validate_bound(Variables, Locked Variables) then
2          V <- select_variable(Variables, LockedVariables)
3          if V=nil then
4             save_best_solution(Variables)
5          else
6             Value <- nil
7             while Value <- select_value(V,Value) & non_expired(V) do
8                label(Variables,LockedVariables) under V=Value
9             end while
10            label(Variables,[V|LockedVariables])
11         end if
12      end if
    end label

    solve(Variables)
       label(Variables,[])
       return saved_best_solution
    end solve
```

Fig. 1. A labeling procedure for solving the minimal perturbation problem.

Figure 1 shows a skeleton of the proposed algorithm that is basically a branch-and-bound algorithm. The algorithm labels variables – assign values to the variables – until a solution is found. First, the algorithm checks whether the current bound is better than the bound of the so far best assignment, if any (row 1). In case of success, the algorithm continues by extending the current partial assignment (rows 2-11). Otherwise, search in this branch stops because the current partial assignment cannot be extended to the best assignment. In row 2, the algorithm selects a variable V to be labeled. If the variable selection fails (row 3), it means there is no variable to be assigned. Thus, the current solution is stored as the best one (row 4) and the current branch is closed. In case of successful variable selection (rows 5-10), the values in the domain of V are tried to be assigned to V (rows 7-8). The algorithm is called recursively in an attempt to assign the rest of the variables (row 8). If it is not possible to select any value for the variable V then the variable is locked and the algorithm continues in labeling the remaining variables (row 10).

In the next sections, we will give details on variable locking and variable expiration, we will describe how to estimate the bound for the optimization algorithm (`validate_bound`), and we will describe the value (`select_value`) and variable (`select_variable`) ordering heuristics used in our solver.

Locked and Expired Variables for Partial Assignments

We have formulated a minimal perturbation problem in such a way that an incomplete assignment could be a solution to the problem. Traditionally, the depth-first search algorithms backtrack to the last assigned variable when all attempts to assign a value to the variable failed. Then they try to find another value for it. Notice that we do not get a locally maximal consistent assignment in such a case because it could still be possible to extend the partial assignment to another non-assigned variable. Therefore, we introduce here the concept of *variable locking*. The variable whose assignment failed is locked and the search algorithm proceeds to the remaining non-assigned variables (row 10). The locked variables still participate in constraint propagation so the above mechanism extends any partial consistent assignment of variables to locally maximal consistent assignment. Notice also that the locking mechanism is local – the variable is locked only in a given search sub-tree. As soon as the algorithm backtracks above the level, where the variable has been locked, the variable is unlocked and it can participate in labeling again.

Recall that our original motivation was to support interactive changes in the problem. It means that the solver returns a solution quickly after any change. Exploring a complete search space of the problem could be hard and a time consuming task. We propose to explore just a part of the search space by applying techniques of LAN Search [18]. The basic idea of LAN (Limited Assignment Number) Search is to restrict the number of attempts to assign a value to the variable by a so called *LAN limit*. This number is maintained separately for each variable, which differentiates LAN Search from other incomplete tree search techniques like Credit Search [3] and Bounded Backtrack Search [9].

The LAN principle is realized by using a counter for each variable. The counter is initialized by the LAN limit. After each assignment of a value to the variable, this counter is decreased. Note that it is possible to assign the same value to the variable in different branches of the search tree and each such attempt is reflected in the counter. When the counter is zero, the variable reached the allowed number of assignments – we say that the *variable expired*.

Let us assume that the time complexity of the search algorithm is defined as a number of attempts to assign a value to a variable. Because different combinations of values should be explored, the worst-case time complexity of the complete search algorithms is exponential expressed by the formula $domain\_size^{number\_of\_variables}$. The LAN limit restricts the number of attempts to assign a value to each variable in the search tree. Thus, the total number of attempts to assign a value to the variables is $lan\_limit*number\_of\_variables$ that is we get a linear worst-case time complexity of the LAN search algorithm. Consequently, the LAN search algorithm does not explore all possible combinations of values and it produces approximate solutions only.

Locked and expired variables do not participate in labeling. Thus, these variables are skipped when selecting a variable to be labeled (`select_variable`), even if they are not assigned yet. Moreover, the variable may expire when already selected for labeling. So, the expiration must be checked during value selection as well (row 7).

Computing Bounds for Optimization

As we mentioned above a minimal perturbation problem is a sort of constraint optimization problems. There are two objective criteria in MPP, namely maximizing the number of assigned variables (called *length*) and minimizing the distance from the initial assignment. Our solver uses the distance function defined as a number of perturbations so the proposed algorithm is looking for the largest consistent assignment and it prefers the assignment with a smaller number of perturbations among the assignments of the same length. Thus the objective function can be described as a lexicographic ordering [maximize length, minimize perturbations].

The core labeling procedure explores locally maximal consistent assignments. When it finds such an assignment, it saves it as a bound (row 4) that is used to prune the rest of the search tree. The pruning is done by checking whether the current partial assignment can be better than the so-far best saved assignment (row 1). To do this comparison, we estimate the maximal length of the assignment and the minimal number of perturbations. In particular, we need an upper estimate of the maximal length and a lower estimate of the number of perturbations. Computing the estimates is included in the function `validate_bound`.

The estimate of the maximal length is simply the number of all variables minus the number of locked variables. Recall, that the algorithm did not succeed to assign a value to the locked variable. Thus, this variable will not participate in a locally maximal consistent assignment. On the other hand, the expired variables may still be part of a locally maximal consistent assignment because the value can be selected for them via constraint propagation.

We estimate the minimal number of perturbations by counting the variables where the initial value is outside their current domain. The initial value for the variable is taken from the initial assignment. If no such value exists – no value is assigned to the variable in the initial assignment – then the variable is always assumed as a perturbation. This is realized by using a dummy initial value for v outside the domain D_v. Note, that there is a constant number of such variables, say K, in the problem so minimizing |W| – the size of the distance set – is equivalent to minimizing |W|+K.

Value and Variable Ordering Heuristics

The proposed algorithm can use existing variable ordering heuristics encoded in the procedure `select_variable`. Only the variables that are neither locked nor expired and that are non-assigned yet can be selected for labeling. We use a standard *first-fail principle* to select among them – the variable with the smallest domain is selected.

The value ordering heuristics (`select_value`) should reflect the goal of search – finding an assignment minimizing the number of perturbations. Therefore, for each value we compute the number of perturbations that will appear in the solution if the value is selected – *added perturbations*. The values with a smaller number of added perturbations are preferred. Computing the number of added perturbations can be done by a problem-specific procedure (like in our tests) or by a problem independent method like singleton consistency (the value is assigned, propagated through the constraints, and the added perturbations are counted).

Experimental Results

We have implemented the proposed labeling procedure in `clpfd` library [2] of SICStus Prolog version 3.10.1. For comparison, we have also implemented a local search procedure in Java, JDK 1.3.1. All presented results were accomplished under Linux on one processor of a PC with a Dual Pentium IV Xeon 2.4 GHz processor with 1 GB of memory. Algorithms were tested using a benchmark problem, called a random placement problem, derived from timetabling problems.

Random Placement Problems

The *random placement problem* (*RPP*) (see http://www.fi.muni.cz/~hanka/rpp/) seeks to place a set of randomly generated rectangles – *objects* – of different sizes into a larger rectangle – *placement area* – in such a way that no two objects overlap and all objects' borders are parallel to the border of the placement area. In addition, a set of allowable placements can be randomly generated for each object. The ratio between the size of the placement area and the total area of all objects is denoted as the *filled area ratio*.

RPP allows us to generate various instances of the problem similar to a trivial timetabling problem. The correspondence is as follows: the object corresponds to a course to be timetabled – the x-coordinate to its time, the y-coordinate to its classroom. For example, a course taking three hours corresponds to an object with dimensions 3×1 (the course should be taught in one classroom only). Each course can be placed only in a classroom of sufficient capacity – we can expect that the classrooms are ordered increasingly in their size so each object will have a lower bound on its y-coordinate.

RPP is modeled as a CSP. Each object is represented by a pair of variables: x-coordinate (*VarX*) and y-coordinate (*VarY*) in the placement area. Each variable is given a domain according to the size of the placement area and in case of y-coordinate also according to its lower bound. The global constraint `disjoint2` [2] over x-coordinates and y-coordinates ensures that the objects will not overlap.

Each object is also represented by a redundant *time-space variable* defined by the constraint *VarXY* #= *VarX* + *VarY* * *SizeX*, where *SizeX* is the size of the placement area in the x-coordinate. Note that there is a one to one mapping between the value of *VarXY* and the pair of values for *VarX* and *VarY* defining it (see Figure 2).

	2	8	9	10	11
VarY	1	4	5	6	7
	0	0	1	2	3
		0	1	2	3
			VarX		

Fig. 2. A value of the time-space variable *VarXY* (in the matrix) is defined uniquely by the positions *VarX* and *VarY* via the formula *VarXY* = *VarX* + *VarY* * *SizeX*, where *SizeX*=4.

We can post the global constraint `all_distinct` over the *VarXY* variables which has a similar role like the `disjoint2` constraint. Using both constraints together brings a higher degree of propagation. However, the main reason for introducing the time-space variables is that they will participate in the labeling procedure. It has the advantage of exact placement of the object by assigning a value to its time-space variable.

Let us now describe how we can generate the changed problem and how we evaluate a solution of the MPP. First, we compute the initial solution using the LAN search algorithm [18]. The changed problem differs from the initial problem by *input perturbations*. An input perturbation means that both x-coordinate and y-coordinate of a rectangle must differ from the initial values, i.e. *VarX#\=InitialX*, *VarY#\=InitialY*. For a single initial problem and for a given number of input perturbations, we can randomly generate various changed problems. In particular, for a given number of input perturbations, we randomly select a set of objects which should have input perturbations. The solution to MPP can be evaluated by the number of *additional perturbations*. They are given by subtraction of the final number of perturbations and the number of input perturbations.

Experiments

We have completed a number of small experiments on RPPs during implementation of the algorithm. RPPs were very helpful as they are simple enough to help in understanding behavior of the algorithm. On the other hand, one can generate much larger RPPs, which allow performing more complex experiments.

The first set of experiments shows performance of the algorithm for different sizes of the random placement problem. We have generated three data sets with 100, 200, and 300 objects, respectively. The objects were generated randomly in the sizes 2×1 to 6×1. Each object has assigned a random lower bound for the y-coordinate which was in average 35% of the y-size of the placement area. These parameters correspond to a real-life course timetabling problem that we are going to solve in the future. Each data set contained fifty problems and the filled area ratio was 80%. The initial solution was computed by the LAN search algorithm. Then we added changes to the solution by forbidding the current position of randomly selected objects. Recall that these changes are called input perturbations. The number of input perturbations was chosen in the range from 0 to 25% of the number of objects. For each problem and for each considered number of the input perturbations, five minimal perturbation problems were generated. During the experiments, the algorithm used a LAN limit equal to 5. We also performed experiments with the limit 10 and 100 but no improvement has been achieved.

Figure 3 shows the relative number of additional perturbations, the relative number of assigned variables, and the CPU time as a function of the number of input perturbations. We use relative numbers of perturbations with respect to the number of objects to compare the results for data sets with a different number of objects.

The top left graph of Figure 3 shows an increase of the number of additional perturbations with the increasing number of input perturbations. This behavior is natural – more changes in the problem definition evoke more additional changes in the solution. We can also see that the number of additional perturbations decreases with the size of the problem which may seem surprising at first. However, recall that the size of the objects remains the same while the size of the area is increasing with the larger number of objects (to keep the filled area ratio 80%). Note that it is easier to place 80 objects into an area of size 100 than to place 8 objects into an area of size 10 which explains the above behavior.

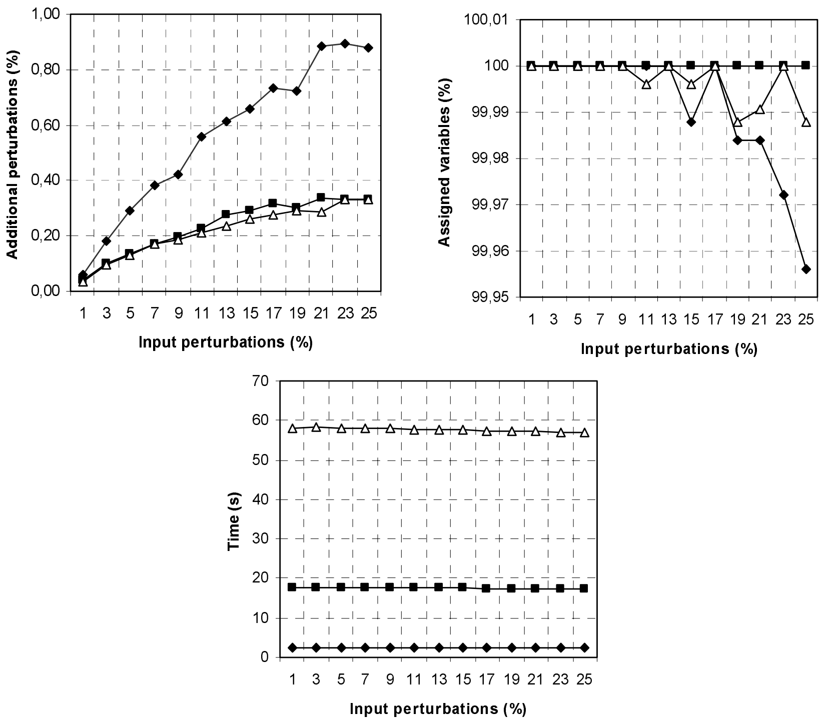
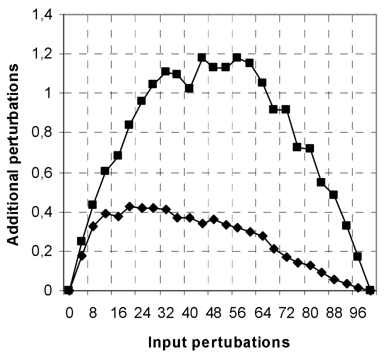


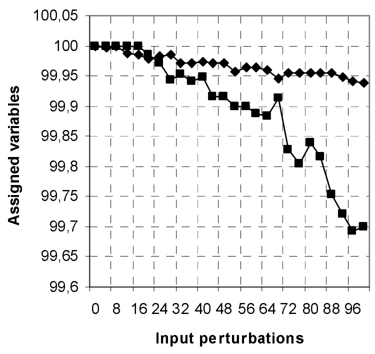
Fig. 3. Performance and quality of the solution for the proposed algorithm on random placement problems with 100 objects (♦), 200 objects (■), and 300 objects (△).

The top right graph of Figure 3 shows a number of assigned variables which is rather large. Actually, all variables were assigned in most of the experiments which is a very promising result. The reason could be that the filled area ratio is 80% which makes the problems under-constrained. Due to time reasons we were not able to perform experiments for over-constrained problems yet.

The bottom graph of Figure 3 shows a non-surprising increase of the running time with larger problems. The increase is non-linear which may seem contradicting with the linear search space explored by the algorithm. However, it is necessary to include the complexity of value ordering heuristics in the total complexity. The complexity of heuristics depends on the number of objects as well as on the number of positions where the objects can be placed.

In the second experiment, we have compared the proposed extension of the branch-and-bound algorithm with a local search algorithm. This local search algorithm is an extended version of the algorithm presented at [13]. It is heuristically oriented to solving RPPs and it was implemented in Java. As before, we used fifty initial problems and five MPPs per an initial problem. We compared the algorithms on the problems consisting of 100 objects and we did the experiments for input perturbations from 0 to 100 with the step 4. Thus 0 to 100% relative input perturbations are covered.





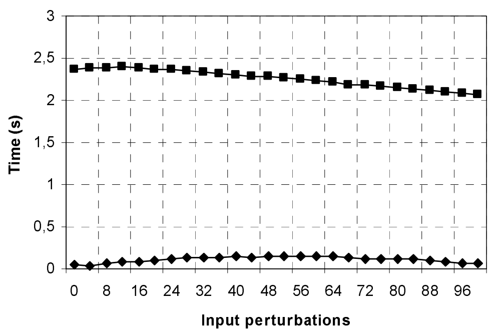


Fig. 4. Comparison of the proposed branch-and-bound algorithm (■) with a problem specific local-search solver (♦).

Figure 4 shows the number of additional perturbations, the number of assigned variables, and the CPU time as a function of the number of input perturbations for both algorithms. We expect the comparison to be the most meaningful for a smaller number of input perturbations (up to about 25%) because this is the area of the highest interest for MPPs. Here the general branch-and-bound algorithm seems to be comparable to the specific problem solver in terms of the solution quality. Still, the dedicated local search algorithm is much faster there.

Conclusions

Including dynamic features to solvers is a recent trend in planning, scheduling, and timetabling. In this paper, we addressed one of these features, namely finding a solution of changed problem that does not differ much from the solution of the original problem – a so called minimal perturbation problem. We proposed a new view of the minimal perturbation problem in the context of constraint satisfaction motivated by a real-life application of university course timetabling. This new framework supports incomplete assignments as a problem solution which is useful to model over-constrained problems as well as hard-to-solve problems. Moreover, this new formulation allows looser changes of the problem formulation like addition and retraction of constraints and variables as well as changes in the variables' domains. We also proposed a new labeling algorithm to solve minimal perturbation problems. This incomplete algorithm has a linear time complexity and it is capable to solve over-constrained and hard problems by looking for locally maximal consistent assignments. The experiments using random placement problems showed that the algorithm is able to find optimal or close to optimal solutions. Next work will concentrate on testing the algorithm in a real-life timetabling application and on improving its real-time performance. We will also explore the possibility to extend the tested local search algorithm for general minimal perturbation problems.

Acknowledgements

This work is supported by the Grant Agency of Czech Republic under the contract No. 201/01/0942 and by Purdue University. Our thanks go to the Supercomputer Center Brno where experiments were conducted. Last but not least we would like to thank to Vlastimil Křápek who implemented an automated tester needed for experimental evaluation.

References

1. Roman Barták, Tomáš Müller, and Hana Rudová. *Minimal Perturbation Problem – A Formal View*. Neural Network World 13(5): 501-511, 2003.
2. Mats Carlsson, Greger Ottosson, and Bjorn Carlson. *An open-ended finite domain constraint solver.* In Programming Languages: Implementations, Logics, and Programming. Springer-Verlag LNCS 1292, 1997.
3. Andrew M. Cheadle, Warwick Harvey, Andrew J. Sadler, Joachim Schimpf, Kish Shen and Mark G. Wallace. *ECLiPSe: An Introduction.* IC-Parc, Imperial College London, Technical Report IC-Parc-03-1, 2003.
4. Rina Dechter and Avi Dechter. *Belief maintenance in dynamic constraint networks.* In Proceedings of AAAI-88, pp. 37-42, 1998.
5. Brian Drabble, Najam-ul Haq. *Dynamic Schedule Management: Lessons from the Air Campaign Planning Domain.* In Pre-proceedings of the Sixth European Conference on Planning (ECP-01), pp. 193-204, 2001.
6. Hani El Sakkout, Thomas Richards, and Mark Wallace. *Minimal Perturbation in Dynamic Scheduling*. In Henry Prade (editor): 13th European Conference on Artificial Intelligence. ECAI-98. John Wiley & Sons, 1998.
7. Hani El Sakkout and Mark Wallace. *Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling*. Constraints 4(5): 359-388. Kluwer Academic Publishers, 2000.
8. Eugene C. Freuder and Richard J. Wallace. *Partial Constraint Satisfaction.* Artificial Intelligence, 58:21-70, 1992.
9. William D. Harvey. *Nonsystematic backtracking search.* PhD thesis, Stanford University, 1995.
10. William D. Harvey and Matthew L. Ginsberg. *Limited discrepancy search*. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, pp. 607-615, 1995.
11. Narendra Jussien and Olivier Lhomme. *Local search with constraint propagation and conflict-based heuristics.* Artificial Intelligence, 139(1):21-45, 2002.

12. Waldemar Kocjan. *Dynamic scheduling: State of the art report*, SICS Technical Report T2002:28, ISSN 100-3154, 2002.
13. Tomáš Müller and Roman Barták. *Interactive Timetabling: Concepts, Techniques, and Practical Results.* In Edmund Burke and Patrick De Causmaecker (eds.): Proceedings of PATAT2002 , Gent, pp. 58-72, 2002.
14. Yongping Ran, Nico Roos, Jaap van den Herik. *Approaches to find a near-minimal change solution for Dynamic CSPs.* Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems, pp 373-387, 2002.
15. Hana Rudová and Keith Murray. *University Course Timetabling with Soft Constraints.* In Edmund Burke and Patrick De Causmaecker (eds.): Practice And Theory of Automated Timetabling IV. Springer-Verlag LNCS 2740, pp. 310-328, 2003.
16. Hana Rudová, *Soft CLP(FD).* In Susan Haller and Ingrid Russell (eds.): Proceedings of the Sixteenth International Florida Artificial Intelligence Symposium, FLAIRS-03, AAAI Press, pp. 202-206, 2003.
17. Gérard Verfaillie, Narendra Jussien. *Dynamic Constraint Solving.* A tutorial including commented bibliography presented at CP 2003, Kinsale.
18. Kamil Veřmiřovský and Hana Rudová, *Limited Assignment Number Search Algorithm.* In Maria Bielikova (ed.): SOFSEM 2002 Student Research Forum, pp. 53-58, 2002.
19. Stefan Voß. *Meta-heuristics: State of the art.* In Alexander Nareyek (ed.): Local search for planning and scheduling: revisited papers. Springer-Verlag LNAI 2148, pp. 1-23, 2001.

Protein Folding in CLP($\mathcal{FD}$) with Empirical Contact Energies

Alessandro Dal Palù[1], Agostino Dovier[1], and Federico Fogolari[2]

[1] Dip. di Matematica e Informatica, Univ. di Udine
Via delle Scienze 206, 33100 Udine, Italy
{dalpalu,dovier}@dimi.uniud.it

[2] Dip. Scientifico-Tecnologico, Univ. di Verona
Strada Le Grazie 15, 37134 Verona, Italy
fogolari@sci.univr.it

Abstract. We present a declarative implementation in Constraint Logic Programming of the Protein Folding Problem, for models based on Face-Centered Cubes. Constraints are either used to encode the problem as a minimization problem or to prune the search space. In particular, we introduce constraints using secondary structure information.

1 Introduction

The *protein folding problem* is the problem of predicting the 3D structure of the protein (its *native conformation*) when the linear sequence of aminoacids identifying the protein is known. It is widely accepted that the native conformation ensures a state of minimum free energy. An energy function associated to a conformation depends on *distances* between any pair of aminoacids and on their *type*. Recently, a new potential matrix able to provide the energy between two aminoacids in *contact* depending on their type has been developed [2]. Thus, the problem can be reduced to that of minimizing this function on the admissible 3D conformations for the protein. This problem (the decision version of it) is proved to be NP-complete [6]. However, the problem, which is of crucial importance in Biology and Biotechnology deserves to be attacked, and we can be encouraged by the non-huge typical length of a protein. Moreover, the fact that proteins fold extremely fast witnesses the existence of a hidden mechanism. Some knowledge about this mechanism surely improves any previous attempt. One of the ingredients of this mechanism could be the fact that particular partial conformations such as helices and sheets are recognizable in most proteins. These structural elements are called *Secondary Structure* of a protein. The high accuracy obtained by secondary structure predictions [12] prompts for inclusion of this prediction for protein folding prediction.

Constraint Programming [9] is a programming methodology well-suited for encoding combinatorial optimization problems. The programmer is allowed to incrementally add new constraints to a first prototype to prune the (typically exponential) space where the solutions to the problem are searched. Constraints are first-order formulas to be interpreted over various possible domains; typically, combinatorial problems are encoded with constraints over *finite* domains.

We use Constraint Logic Programming over finite domains in SICStus Prolog [18] to deal with the Protein Folding Problem. We first encode the problem as a minimization problem and then we add constraints obtained by secondary structure prediction to reach acceptable computation time. We study the effectiveness of the method on some small proteins whose structure is known.

This work is a continuation of a preliminary proposal [7]. In [7] the indexes for the angles between consecutive residues were referred to a global coordinate system reducing the orientational degrees of freedom (and consequently, admissible results). In this work we use local coordinate systems to define torsional angles. Moreover we developed a new method to dynamically prune the search tree, based on the analysis of the contacts between the aminoacids during the folding process. The results we obtained allow us to effectively predict proteins up to 50 aminoacids, and they are improved by introducing new constraints into the code and by developing new *labeling* strategies. In Table 1 we compare the previous results in [7] to the ones in this work. For small proteins, e.g. 1LE3 (16 aminoacids), we have a speedup of 75 times for the running times and the protein energy is improved by more than 3 times. For longer proteins, e.g. 1ED0 (46 aminoacids), the speedup is 270 and the energy is slightly improved. Note that for the protein 1ED0, we reach the physical limit of memory used on our machine, while the computation time is still acceptable. We also compare in Table 2 our results with results obtained by the well-known HMMSTR/Rosetta Prediction System [15].

2 Related Work

There exists a wide bibliography concerning the Protein Folding Problem. For detailed and up-to-date reviews, see [4,14]. All prediction methods make use of statistical information available from the more than 20000 structures deposited in the Protein Data Bank (PDB) [3]. The correct fold for a new sequence can be obtained when *homology* (sequence similarity) is detected with a sequence for which the structure is available. Another approach tries to superimpose (*thread*) a chain on a known structure and to evaluate the plausibility of the fold. *Ab-initio* methods, instead, try to find the native conformation without a direct reference to a structural model.

As far as the constraint community is concerned, a subproblem of the whole problem, based on the *HP-model* is studied and successfully solved for proteins of length 30–40 in [1] and of length 160 in [21]. This model divides aminoacids into two classes (H and P) and the problem reduces to that of finding the conformation that maximizes the contacts of H's. This approach is very interesting, but the high level abstraction of the model does not ensure that the result is the native conformation; in particular, the local sub-conformations of the form of α-helices or β-strands (cf. Sect. 3) are often lost. Let us observe, however, that predictions based on the HP-model are typically a part of a more complex prediction system, and they have to be seen as a preprocessing step, or as a method for calculating general properties of proteins. In [17] the related side-chain problem is studied inside the CLP framework. To give a practical idea about the differences among models and experimental results, we depict the real structure of the protein 1ZDD (Fig. 1), and the ones predicted with the HP-model (Fig. 3) and with our empirical contact energies model (Fig. 2). Note that the HP-model usually returns a compact protein, where the H aminoacids are constrained close to the center of the molecule. Moreover, in HP-foldings on face-centered cubic lattices ($\mathcal{FCC}$—see Section 3.2), often bend angles of 60° are predicted, while this is not common in nature. For instance, for 1ZDD the HP-model outputs a protein with an RMSD (Root Mean Square Deviation) of 8.2 Å w.r.t. the native structure of Fig. 1, while our prediction reaches a lower RMSD of 6.0 Å.

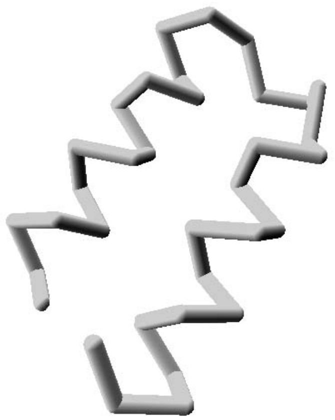


Fig. 1. Protein 1ZDD from PDB

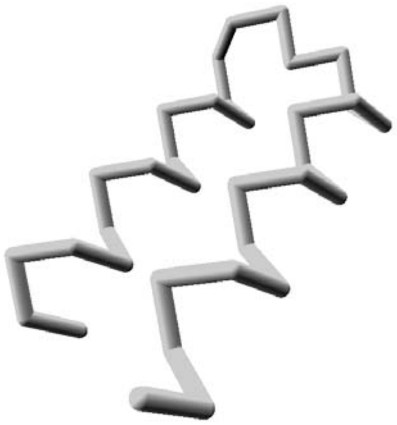


Fig. 2. Protein 1ZDD, our model, 6.0 Å RMSD error

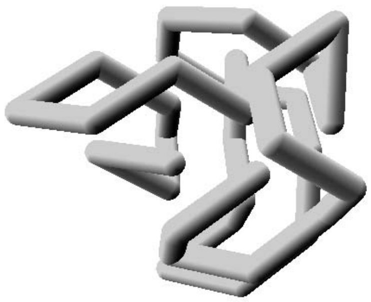


Fig. 3. Protein 1ZDD, HP-model, 8.2 Å RMSD error

3 The Protein Folding Problem

The *Primary* structure of a protein is a sequence of linked units (or *residues*) of reasonable length. Each residue is an aminoacid, from a set $\mathcal{A}$ of 20 types:

Alanine (A)	**Cys**teine (C)	**Asp**artic Acid (D)	**Glu**tamic Acid (E)
Phenylalanine (F)	**Gly**cine (G)	**His**tidine (H)	**I**so**le**ucine (I)
Lysine (K)	**Leu**cine (L)	**Met**hionine (M)	**As**paragi**n**e (N)
Proline (P)	**Gl**utami**n**e (Q)	**Arg**inine (R)	**Ser**ine (S)
Threonine (T)	**Val**ine (V)	**Tryp**tophan (W)	**Tyr**osine (Y)

Applying statistical methods on structures obtained by X-Rays and NMR experiments, a matrix that points out the energy associated to a pair of non consecutive aminoacids when they are in contact has been developed [10,2]. With $\mathsf{Pot}(x, y)$ we denote the value regarding aminoacids x and y (the order is immaterial).

Native conformations are largely built from *Secondary Structure elements* (i.e., helices and sheets) often arranged in defined motifs. α-helices are constituted by 5 to 40 contiguous residues arranged in a regular right-handed helix with 3.6 residues per turn. β-sheets are constituted by extended strands of 5 to 10 residues. Each strand is made of contiguous residues, but strands participating in the same sheet are not necessarily contiguous in sequence. There are algorithms based on neural networks that can predict with high accuracy (75% [4]) the secondary structure of a protein. Another important structural feature of proteins is the capability of cysteine residues of covalently binding through their sulphur atoms, thus forming disulfide bridges, which impose important *contact* constraints (also known as ssbonds). This kind of information is often available, either by experiments or predictions.

3.1 Problem Definition

Several models are proposed for reasoning about the 3D properties of proteins, basically concerning the admissible spatial positions of each aminoacid (also called the *Tertiary Structure* of the protein). Given a sequence $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, the position $\omega(i)$ of a point representing each aminoacid s_i is a triple $\langle x_i, y_i, z_i \rangle$ [1]. Moreover, x_i, y_i, and z_i can be real numbers (in models in which proteins are left free to take any position) or integer numbers (in models where aminoacids can take a finite number of positions of a suitable lattice). We call $\mathcal{D}$ the set of admissible points.

We use the predicate next stating that two positions are admissible consecutive positions for two aminoacids that appear consecutively in the sequence. next requires as parameters the two points, the domain $\mathcal{D}$, and the whole sequence ω of positions, since in some lattices the fact that two points are in sequence depends also on other points (for instance, the previous one).

[1] Precisely, $\langle x_i, y_i, z_i \rangle$ are the coordinates of the center of a particular carbon atom of the aminoacid s_i, known as C_α. C_α atoms define the trace of the protein.

It is assumed that a fixed distance separates two consecutive aminoacids. We also employ the binary predicate $\mathsf{contact}$: two non-consecutive aminoacids s_i and s_j in the position $\omega(i)$ and $\omega(j)$ are in contact (in this case we write $\mathsf{contact}(\omega(i), \omega(j), \mathcal{D})$) when their distance is less than a certain value. Lattice models simplify the definition of the predicates next and $\mathsf{contact}$. The following definition of the protein folding problem is general enough to apply to several models.

Let $\mathcal{P}$ be a set of admissible positions. Given a sequence $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, a *folding* of S is a function $\omega : \{1, \dots, n\} \longrightarrow \mathcal{D}$ such that:

1. $\mathsf{next}(\omega(i), \omega(i+1), \omega, \mathcal{D})$ for $i = 1, \dots, n-1$, and
2. $\omega(i) \neq \omega(j)$ for $i \neq j$ (in other words, ω defines self-avoiding walks).

The *protein folding problem* is the problem of finding the folding ω of S such that the following energy is minimized:

$$E(\omega) = \sum_{\substack{1 \le i < n \\ i+2 \le j \le n}} \mathsf{contact}(\omega(i), \omega(j), \mathcal{D}) \mathsf{Pot}(s_i, s_j)$$

where the function $\mathsf{contact}$ is a boolean function defined using the model $\mathcal{D}$.

3.2 Lattice Models

Lattice models have long been used for protein structure prediction [13]. A possible model is a lattice whose points are a subset of $\mathbb{N}^3$. Nevertheless, the only angles between three consecutive aminoacids allowed in this case are 90° and 180°. This lattice is too rigid for a realistic modelization of the problem.

The $\mathcal{FCC}$ model. In [11] it is shown that the *Face-Centered Cubic Lattice* ($\mathcal{FCC}$) model is a well-suited, realistic model for 3D conformations of proteins. The model is based on cubes of size 2, but the central point of each face is also admitted (together with the vertices). Thus, the domain $\mathcal{D}$ consists in a set of triples $\langle x, y, z \rangle$ where $x, y, z \in \mathbb{N}$. Points at Euclidean distance $\sqrt{2}$ are connected; their distance is said *lattice unit*. Observe that for connected points i and j it holds that $|x_i - x_j| + |y_i - y_j| + |z_i - z_j| = 2$.

In this way each point is adjacent to 12 neighboring points. The angle between three residues may therefore assume values 60°, 90°, 120°, and 180°. Steric and energetic constraints in proteins make values 60° and 180° unfeasible. Therefore only links at 90° and 120° will be retained [19,8]. No similar restriction exists on torsional angles among four adjacent residues. We can therefore define the function next as follows: $\mathsf{next}(\omega(i), \omega(i+1), \omega, \mathcal{D})$ holds iff

- $|x_i - x_j| \in \{0, 1\}, |y_i - y_j| \in \{0, 1\}, |z_i - z_j| \in \{0, 1\}$,
- $|x_i - x_j| + |y_i - y_j| + |z_i - z_j| = 2$ [2], and, moreover,
- let $\boldsymbol{v}_{i-1,i}$ and $\boldsymbol{v}_{i,i+1}$ be the vectors connecting the points $i-1, i$ and $i, i+1$, respectively, computed w.r.t. the point $\langle x_i, y_i, z_i \rangle$. To impose that the angle between them can only be of 90° and 120°, we use the scalar product between these two vectors: $\boldsymbol{v}_{i-1,i} \boldsymbol{v}_{i,i+1} = |\boldsymbol{v}_{i-1,i}||\boldsymbol{v}_{i,i+1}| \cos(\theta)$. Thus, since $|\boldsymbol{v}_{i-1,i}| = |\boldsymbol{v}_{i,i+1}| = \sqrt{2}$ we only need to impose that: $\boldsymbol{v}_{i-1,i} \boldsymbol{v}_{i,i+1} \in \{-1, 0\}$.

[2] Or, equivalently, $(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 = 2$.

Contact. A *contact* is defined among two non adjacent residues when their separation is two lattice units (i.e., viewing the lattice as a graph whose edges connect successive points, they are connected by a path of length 2). The more natural choice of a single lattice unit is ruled out because it does not take into account aminoacid steric hindrance. Physically, a lattice unit corresponds to 3.8 Å, which is roughly the van der Waals contact distance between two carbon atoms. Recent results (see, e.g., [2]) show that a contact between two residues, when represented only by their C_α atoms, is optimally defined for C_α-C_α distances shorter than 6.4 Å [3]. Our choice of 2 lattice units accounts for all contacts of $3.8 \times \sqrt{2} = 5.4$ Å can therefore be considered a good approximation. In order to avoid the possibility for two non consecutive residues to have the same distance as two consecutive ones (which is a non frequent situation), we also impose the constraint that two non consecutive residues must be separated by more than one lattice units. This is achieved by adding, for each pair of non-consecutive points i and j, the constraint:

$$(|x_i - x_j| > 1) \vee (|y_i - y_j| > 1) \vee (|z_i - z_j| > 1) \vee (|x_i - x_j| + |y_i - y_j| + |z_i - z_j| > 2)$$

With this further constraints, contact can be defined as follows:

– $\mathsf{contact}(\omega(i), \omega(j), \mathcal{D})$ holds if and only if (cf. also footnote 2)

$$|x_i - x_j| + |y_i - y_j| + |z_i - z_j| = 2\,.$$

4 Definition in *CLP*($\mathcal{FD}$)

In this section we describe the main predicates used to implement declaratively the protein folding problem. We use the library clpfd of SICStus PROLOG 3.10 [5]. Complete code and other related material can be found in:
`http://www.dimi.uniud.it/~dovier/PF/pf_clp.html`.
The main clause is a classical *Constrain & Generate* [9] clause of the form:

```
fcc_pf(ID):-
    protein(ID, Primary, Secondary),
    constrain(Primary, Secondary, Indexes, Tertiary, Energy, Matrix),
    labeling(Primary, Secondary, Indexes, Tertiary, Energy, Matrix),
    print_results(Primary, Tertiary_flat, Energy).
```

The `protein` predicate, extensionally defined in an auxiliary file `data.pl`, allows us to access to the `Primary` and `Secondary` structures of a protein given its name. For instance,

```
protein(ID, Primary, Secondary):-
    ID = '1KVG', Primary = [s,c,h,f,g,p,l,g,w,v,c,k],
    Secondary = [ ssbond(2,11), strand(2,4), strand(9,11)].
```

[3] The number is obtained as the sum of the radius of the two C_α carbon atoms we are dealing with (2×1.9 Å) and the value of 2.6 Å empirically determined in [2].

The constrain predicate deterministically adds the (finite domain) constraints for the variables involved, while labeling looks for the solution in the search space. print_results prints the output in a format suitable for validating the results with biological software (cf. Section 5). Tertiary is the output list of positions (flat list of triples of integers) of the various aminoacids (the conformation) and Energy is the output value of energy associated to it (the values of the Table developed in [2] are multiplied by 1000 in order to deal with integer values). Indexes is an auxiliary variable that we discuss, together with the variable Secondary, in the next subsection. Just as an example, consider a possible computation:

```
| ?- fcc_pf('1KVG', Energy).
 Global time: 2,365s
 1 s 12 12 12
 2 c 13 13 12
 ...
 12 k 13 11 10
 Energy = -15394
```

The predicate constrain is defined as follows:

```
constrain(Primary, Secondary, Indexes, Tertiary, Energy, Matrix):-

  length(Primary,N),
  generate_tertiary(N, Primary, Tertiary),
  domain_bounds(N, Tertiary),
  avoid_symmetries(Indexes, Tertiary),
  avoid_self_loops(Tertiary,N),
  next_constraints(Tertiary),
  distance_constraints(Tertiary),

  generate_indexes(N, Indexes),
  secondary_info(Secondary, Indexes, Tertiary),
  indexes_to_coordinates(Indexes, Tertiary),

  energy_constraints(Primary, Secondary, Tertiary, Indexes, Energy,
                     Matrix).
```

generate_tertiary generates the list Tertiary $= [X_1, Y_1, Z_1, \ldots, X_N, Y_N, Z_N]$ of $3N$ variables, where N is the length of the input list Primary. The first group of predicates adds constraints to those variables. domain_bounds bounds to $0 \ldots 2*N$ all the variables X_i, Y_i, Z_i. Moreover, exploiting some lattice properties (as done in [1]) we also force $X_i + Y_i + Z_i$ to be even. avoid_symmetries removes redundant admissible conformations equivalent to others modulo some symmetries and/or rotations. The predicate assigns immediately three consecutive aminoacids positions (in the Tertiary list). avoid_self_loops forces all triples to be distinct (we use the built-in predicate all_different on the list

$[B_1, \ldots, B_n]$ where B_i `#=` $(X_i * P * P) + (Y_i * P) + Z_i$, for a suitable value P). `next_constraints` imposes that $[X_i, Y_i, Z_i]$ and $[X_{i+1}, Y_{i+1}, Z_{i+1}]$ are adjacent points in the lattice (see Section 3.2). `distance_constraints` forces two non-consecutive points to be at a lattice distance of at least 2.

4.1 Use of Secondary Structure Information

The three predicates `generate_indexes`, `secondary_info`, and `indexes_to_coordinates` concern with the encoding of Secondary Structure Information in the program. We are interested in secondary structure information of the kind:

`helix`(i,j)**:** elements $i, i+1, \ldots, j$ of the input sequence form an α-helix,
`strand`(i,j)**:** elements $i, i+1, \ldots, j$ are in a β-strand,
`ssbond`(i,j)**:** presence of a disulfide bridge between element number i and j.

This information is contained in the list `Secondary`; for our tests, we have picked it from the Protein Data Bank [3]. It is of course possible to modify the code so as to use automatically the prediction given by the program PhD [12] and/or other programs.

In the $\mathcal{FCC}$ model it is useful to adopt also another representation for imposing constraints regarding helices and strands. Given two points, only 6 possible directions for the next point are allowed. Thus, with a list of $N - 3$ numbers ranging from 0 to 5 (called *Indexes*) we can precisely identify the space position of each aminoacid. To obtain this dual description, we define a local relationship between a set of four consecutive positions p_1, p_2, p_3, p_4 and the index associated to the fourth of them p_4. Let us define $\boldsymbol{r}_{12} = p_2 - p_1, \boldsymbol{r}_{23} = p_3 - p_2, \boldsymbol{r}_{34} = p_4 - p_3$. The idea is that, given $\boldsymbol{r}_{12}$ and $\boldsymbol{r}_{23}$, it is possible to associate a unique code to the direction $\boldsymbol{r}_{34}$. Thus, the dual representation allows us to specify a subsequence conformation not depending on other aminoacids not yet placed.

In detail, $\boldsymbol{r}_{12}$ and $\boldsymbol{r}_{23}$ define a plane π. Let us call σ the plane orthogonal to π and to $\boldsymbol{r}_{23}$. We also define $\boldsymbol{i}$ as the normalized projection of $\boldsymbol{r}_{12}$ on σ and $\boldsymbol{j}$ as the normalized vector product of $\boldsymbol{i}$ and $\boldsymbol{r}_{23}$. The three vectors $\boldsymbol{i}, \boldsymbol{r}_{23}, \boldsymbol{j}$ are an orthogonal base for $\mathbb{R}^3$. Note that the plane σ is defined by $\boldsymbol{i}$ and $\boldsymbol{j}$. The normalized projection of $\boldsymbol{r}_{34}$ on σ matches exactly one of the following:

0 is equal to $\boldsymbol{i}$	**3** is equal to -$\boldsymbol{i}$
1 falls in the first quadrant	**4** falls in the third quadrant
2 falls in the second quadrant	**5** falls in the fourth quadrant

With these conventions, helices are sequences of the form: $5, 5, 5, \ldots$ while β-strands are associated to sequences of the form: $3, 3, 3, \ldots$. The predicate `secondary_info` adds the constraints concerning the secondary structure information. In particular, `helix`(i,j) forces the variables of `Indexes` associated to elements $i, i+1, \ldots, j$ to have values 5 and `strand`(i,j) forces the variables of `Indexes` associated to elements $i, i+1, \ldots, j$ to have value 3. `ssbond`(i,j), introduces a different type of constraint: $|\mathtt{X_i} - \mathtt{X_j}| + |\mathtt{Y_i} - \mathtt{Y_j}| + |\mathtt{Z_i} - \mathtt{Z_j}|$ `#=< 6`.

The predicate `indexes_to_coordinates` relates the constraints on the indexes to the variables denoting positions. The definition is rather technical and we omit it here due to lack of space. As anticipated in the Introduction, we improve the definition in [7], where a global system of coordinates was used. The predicate imposes bidirectional constraints, so that it is possible for the solver to propagate the domain changes in index variables to the corresponding tertiary variables and vice versa. To achieve this, we define six constraints of the form `Code #=` θ `#<=>` $\texttt{C}_\theta$, where $0 \leq \theta \leq 5$. The bidirectionality is guaranteed by the presence of the `#<=>` operator and by the only use of FD constraints on variables. The C_θ formula is a compact and optimized collection of the distinct configurations associated to the index code θ. Each θ can be obtained from one of 3 different classes of conformations easily described with simple vector algebra. This allows us to cluster the number of explicit cases in the predicate, avoiding a single complex formula.

4.2 Contact Management and `energy_constraints`

As described in section 3.2, each pair A_i and A_j of aminoacids are in contact if a path of 2 lattice units links them. Considering the constraints imposed (i.e., `next` and `non_next`), this relationship can be reformulated as follows:

$$\texttt{contact}(A_i, A_j) \Leftrightarrow 2 = |X_i - X_j| + |Y_i - Y_j| + |Z_i - Z_j|$$

The contact information is maintained in a symmetric matrix M, such that $M[i,j]$ is the energy contribution provided by each pair of aminoacids A_i, A_j. The following code constrain the contact contribution to the matrix element:

```
table(A_i,A_j, Pot),
C in {0,Pot},
2 #= abs(X_i- X_j) + abs(Y_i - Y_j) + abs(Z_i - Z_j)#<=> M[i,j] #= Pot.
```

where `table` reports the value $\mathsf{Pot}(A_i, A_j)$ as computed in [2].

The optimal folding is reached when the sum of M elements is minimal. During the *labeling* phase, the information stored in M is used to control the minimization process and to cut the search tree.

In the implementation of the contact matrix, we identify some heuristics to determine, for every element $M[i,j]$, if it is possible to avoid to build the contact constraint and/or to impose that the element is not representing a contact ($M[i,j] = 0$). Since the Energy to be minimized is strictly connected to M, it is preferable to simplify the matrix to reduce the number of non-ground variables bounded to the tertiary structure.

The constraints applied to M, for each i, j, are:

```
constrain_contact(Matrix,Size,i,j,A1,A2,[X1,Y1,Z1],[X2,Y2,Z2],Segm),
constrain_min_dist(Matrix,Size,i,j).
```

where `Segm` represents the known set of contiguous ground indexes.

In `constrain_contact` the contact constraint described above is applied to those indexes i, j that are *not* included in a contiguous set of ground indexes (e.g. a `helix`(h_1, h_2), with $h_1 \leq i \leq j \leq h_2$). In these cases, in fact, the contact contribution is not taken into account because it is constant for each conformation, thus $M[i,j]$ is directly set to 0 and no constraints are added on the Tertiary structure for A_i and A_j.

To reduce the search space it is possible to ignore the contact contribution for all pairs A_i, A_j such that $j < i + k$, where k is a parameter. In those cases, the constraint is not applied and $M[i,j] = 0$ without any effect on the Tertiary structure. For example, if $k = 3$ then the problem is approximated so that all contacts for aminoacids at distance less than 3 in the Primary sequence are not considered. This choice is useful when looking only for contributions between aminoacids far apart in the chain.

`constrain_min_dist` is an additional heuristic that imposes the constraints $M[i,j] = 0$ for $j < i + \ell$ and $j > i + 2$. The parameter ℓ states that no contacts occur between aminoacids closer than ℓ positions on the Primary sequence. In this case, we impose some '0' to the matrix and if the element $M[i,j]$ has been constrained by `constrain_contact`, this has the effect to disallow the distance of 2 between A_i and A_j. The $\mathcal{FCC}$ lattice structure allows contacts between aminoacid such that $j = i + 2$, according to our definition, but a contact with $j = i + 3$ in the $\mathcal{FCC}$ lattice is associated to a conformation with angles that are not present in nature. Therefore we set $\ell = 4$.

We now describe how the matrix is used to compute the Energy. As said before, the Energy is the sum of M elements. When minimizing Energy it is important to have the least number of bounded variables to Energy. Considering the simple sum of all elements of the matrix is time consuming. Our idea is to extract from M a `PotentialList`, which is the set of non ground elements of M and add the constraint:

```
sum(PotentialList, #=, Energy).
```

Note that the constraints imposed by the predicates described above, combined to the constraint propagation, have as consequence that many elements in M are set and/or predicted to be ground before the beginning of the labeling phase. Thus the `PotentialList` extracted becomes smaller and easier to be managed by the solver.

4.3 Finding Solutions

In the *generate* phase we introduce some heuristics for pruning the search tree.

The size of the problem is such that the constraints imposed cannot be managed by the solver at once. The first heuristic we present (*subsequencing*) aims to identify those regions (*subsequences*) on the protein to be folded first and build a folding based on the already folded areas. Note that a subsequence can be easily identified by the set of indexes associated to it and it is independent from 3D orientation. A sequence of indexes contains runs of variables and ground numbers (regions already folded). We want to select the region containing the smallest ratio of variables over ground indexes (most known subsequence), preferring the ones that include a ssbond. If in the selected subsequence there are too many variables (e.g., more than 10), we cut the subsequence in order to obtain only 10 free variables. The folding of a subsequence is called a *phase*. For small proteins it is possible to avoid the subsequencing and Table 1 reports this choice in the Notes column.

In order to perform a further pruning to the search tree, we add a local labeling technique to the built-in `labeling` predicate. After each instantiation of t variables in a phase, we collect the best known ground admissible solution, its energy and its associated potential matrix. The idea is to compare the current status to history and decide if it is reasonable to cut the search tree. We consider the ground set of elements in the current potential matrix, `CurrentGround`, and compare them to the corresponding set associated to the best solution found, `BestGround` (see Fig. 4). A first manner to cut the search, is to require that `CurrentGround` cannot be worse than half of `BestGround`, as shown in the code:

```
best_solution(BestEnergy),
best_potentialList(BestPL),
sumgrounds( PotentialList, BestPL, CurrentGround, BestGround),
CoefS1 is 0.5,
S1  is integer( CoefS1 * BestGround),
CurrentGround  < S1,
```

This heuristic is not so effective, because it marks as admissible many branches of the search tree that contain worse energies than the ones already computed. On the other hand, it is not possible to increase the coefficient `CoefS1`, because it would exclude too many feasible solutions at the beginning, when a small ground matrix area is considered.

An improved version assigns the coefficient depending on the ground area of the matrix:

```
term_variables(PotentialList,PlV),
length(PlV,LPotVar),
length(PotentialList,LPotTot),
CoefS1 is (0.5 * (LPotVar) + 1.1 * (LPotTot-LPotVar)) / (LTotPot),
```

where `term_variables` is a built–in predicate to extract in constant time the list of non ground variables.

In this version, for a small ground area we allow to generate lower energies (0.5 compared to the best known) with the hope that this choice will pay off in the rest of the matrix to be discovered. When the protein is almost folded, the coefficient becomes higher, and consequently, we cut many solutions that are clearly not improving the local minimum, at that moment.

Note that this technique allows to control at the same time the weakness of the pruning when the protein is almost not folded and which is the ratio for improving ground admissible solutions.

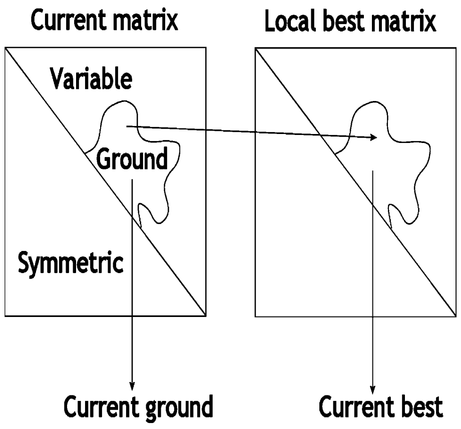


Fig. 4. Contact Matrices

4.4 Optimized Constraints

We present here some optimized constraint used in our code. The fastest `non_next` predicate (used by `distance_constraints`) is:

```
Dx #= (X1-X2)*(X1-X2),
Dy #= (Y1-Y2)*(Y1-Y2),
Dz #= (Z1-Z2)*(Z1-Z2),
Dx+Dy+Dz #>2.
```

compared to

```
Dx #= abs(X1-X2)/2,
Dy #= abs(Y1-Y2)/2,
Dz #= abs(Z1-Z2)/2,
Dx+Dy+Dz #>=1 .
```

and the slowest

```
DX #= abs(X_i - X_j),
DY #= abs(Y_i - Y_j),
DZ #= abs(Z_i - Z_j),
DX #> 1 #\/
DY #> 1 #\/
DZ #> 1 #\/
DX + DY + DZ #> 2.
```

The first one is taking 25% time less than the last one.

We use the `avoid_symmetries` predicate to narrow the domains. As said before this predicate fixes the 3D positions for three aminoacids. Since every set of 3 consecutive aminoacids can be chosen, we instantiate the beginning of a maximal run of ground indexes in the protein, so that the propagation computes automatically the tertiary structure for the whole run. With this choice the minimization process deals with less variables and the performances are increased (e.g. before the labeling phase, the Tertiary list has less variables and their domains are narrowed, the potential matrix contains '0' generated by propagation).

We notice that in proteins containing more than one helix or strand, pairs of subsequences of known secondary structure are often in contact in the optimal conformation. We wrote the predicate `constrain_secondary_contact` to describe that the subsequences A_i, A_j and A_k, A_l are spatially close. The detection of possible set of indexes is not yet an automatic procedure. We did some simple tests and we noticed the positive effects of the predicate on the search process.

5 Experimental Results

We have tested the program using the same set of proteins used in [7][4]. Computational results are reported in Table 1, where "b" stands for ssbond, "s" for strand, and "h" for helix.

In the protein model systems 1LE3, 1PG1, and 1ZDD terminal protecting groups have been neglected. For the protein 1ED0, folding with subsequences is producing an Insufficient memory error.

The computations require much less time than [7]. For a safe comparison, we have divided by a factor of 2 the values of [7] to compute the values of the column Old Times, since the computation for [7] has been made with an old 500MHz processor. Moreover, energies found are on average much lower. We noticed that the energy computed, though, is still not completely related to the quality of the prediction. As sketched in the previous section, the code in [7] forced parallelism in secondary structure elements, while the present implementation samples more (less compact) conformations. The improvement in the present implementation makes the inadequacy of lattice contact definition apparent. As said in Section 3.2, the present definition corresponds to a contact distance of ~ 5.4 Å while much larger distances have been used by other authors [20].

The C_α chain of the protein 1E0M as stored in the PDB and the same protein as computed by the Prolog program described in this paper are depicted in Figure 5 and in Figure 6, respectively[5].

We conclude the section comparing the results of our prediction with those returned by the well-known HMMSTR/Rosetta Prediction System [15]. This program does not use a lattice as underlying model: aminoacids are free to take any position in $\mathbb{R}^3$. For a fair comparison, we have used it as an *ab-initio* predictor (precisely, we have disabled the *homology* and *psi-blast* options). Times are obtained from the result files, but it is not clear to which machine/CPU occupation they are refered. Results are reported in Table 2. Let us observe that execution times and RMSD are comparable.

[4] We have used SICStus PROLOG 3.10.0 [18] and a PC AMD Duron 1000 MHz.

[5] The images are obtained with Accelrys ViewerLite 5.0 `http://www.accelrys.com`.

Table 1. Experimental Results

Name	N	Secondary Info	Time	Old Times	En	Old En	Notes
1LE0	12	[s(2,4),s(9,11)]	4s.	1m.15s.	-9040	-4085	
1KVG	12	[b(2,11),s(2,4), s(9,11)]	2s.	1m.15s.	-15394	-6247	
1LE3	16	[s(2,6),s(11,15)]	5s.	6m.15s.	-13653	-4338	
1EDP	17	[b(1,15),b(3,11), h(9,15)]	26s.	16m.45s.	-22249	-19416	no subsequencing
1PG1	18	[b(6,15),b(8,13), s(4,9),s(12,17)]	0.8s.	13s.	-12027	-2907	
1ZDD	34	[b(5,34),h(3,13), h(20,33)]	41s.	23m.30s.	-9975	-19843	sec. contact 3,13 26,33 ssbond 4
1VII	36	[h(4,8),h(15,18), h(23,32)]	6m.56s.	45m.30s.	-25265	-24725	Coef. 0.7 1.1, tertiary ground at 15
1E0M	37	[s(7,12),s(18,22), s(27,29)]	9m.45s.	9h.38m.	-27594	-21944	
2GP8	40	[h(6,21),h(26,38)]	9m.0s.	3h.18m.	-15040	-12361	sec. contact 26,38 6,21 no subsequencing
1ED0	46	[b(3,40),b(4,32), b(16,26),h(7,18), h(23,30),s(2,4), s(33,34)]	2m.33s.	8h.55m.	-32743	-29916	First subs. coeff: 0.9 1.2 Insuf. mem. error Resumed with first folded subs. coeff: 0.8 1.1
1ENH	54	[h(8,20),h(26,36), h(40,52),s(22,23)]	12m.59s.	1h.42m.	-20650	-24859	Coef. 0.7 1.1 sec. contact 8,20 26,36

Table 2. Comparisons with Rosetta predictions

Name	N	Our Time	Our RMSD	Rosetta Time	Rosetta RMSD
1zdd	34	0m.41s.	6.0	5m.35s.	3.5
1vii	36	6m.56s.	7.9 / 6.9 (4–32)	5m.35s.	4.2
1e0m	37	9m.45s.	7.3 / 4.2 (7–22)	6m.35s.	7.7
2gp8	40	9m.0s.	4.1	6m.35s.	6.4
1ed0	46	2m.33s.	8.7 / 4.7 (7–30)	7m.23s.	8.9
1enh	54	12m.59s.	8.2 / 6.6 (8–52) / 4.0 (8–36)	8m.35s.	7.5

6 Future Work and Conclusions

In this paper we have presented the protein folding problem as a minimization problem using a recently developed table of potentials. We have implemented the code in $CLP(\mathcal{FD})$ and tested it over a sample set of proteins. The encouraging results obtained (that improve those obtained in a preliminary proposal) suggest us how to enhance the heuristic phase with the introduction of more sophisticated techniques for handling substrings already computed as high level units. In the next future we are planning to implement centroids of sidechains [17], to improve the RMSD errors, using the same lattice, to redefine the contact in order to match more closely the contact energy definition and to study an automatic procedure to suggest contacts between secondary structure subsequences.

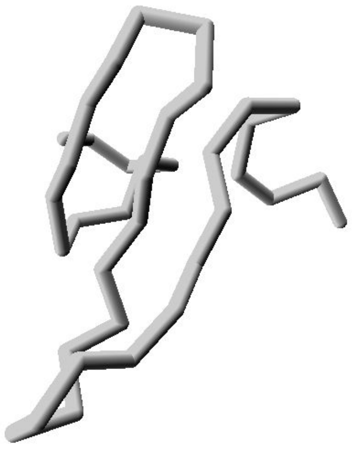


Fig. 5. Protein 1E0M (PDB)

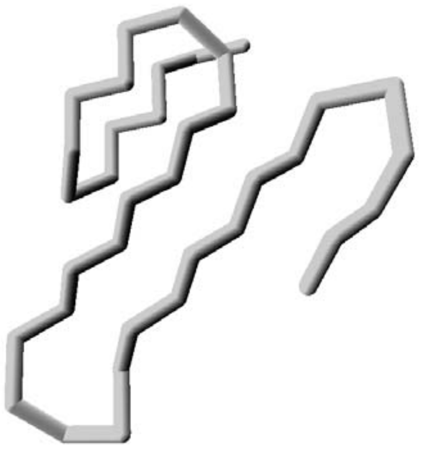


Fig. 6. Protein 1E0M computed

Acknowledgments

We thank the anonymous referees for their useful comments. The work is partially supported by MIUR Projects: *Ragionamento su aggregati e numeri a supporto della programmazione e relative verifiche* and *Verifica di sistemi reattivi basata su vincoli (COVER)* and by the GNCS-INDAM Project: *Metodi Matematici ed Algoritmici per l'Analisi di Sequenze di Nucleotidi e Amminoacidi.* Alessandro Dal Palù was partially supported by ERCIM grant.

References

1. R. Backofen. The protein structure prediction problem: A constraint optimization approach using a new lower bound. *Constraints*, 6(2/3):223–255, 2001.
2. M. Berrera, H. Molinari, and F. Fogolari. Amino acid empirical contact energy definitions for fold recognition in the space of contact maps. *BMC Bioinformatics* 4:8 (2003).
3. H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Research* 28:235–242 (2000). `http://www.rcsb.org/pdb/`.
4. R. Bonneau and D. Baker. Ab initio protein structure prediction: progress and prospects. *Annual Review of Biophysics and Biomolecular Structure*, 30:173–89, 2001.
5. M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In H. Glaser, P. H. Hartel, and H. Kuchen eds. Proc. of *9th PLILP*. Vol. 1292 of LNCS, pp. 191–206, 1997.
6. P. Crescenzi, D. Goldman, C. Papadimitrou, A. Piccolboni, and M. Yannakakis. On the complexity of protein folding. In *Proc. of STOC*, pages 597–603, 1998.
7. A. Dovier, M. Burato, and F. Fogolari. Using Secondary Structure Information for Protein Folding in $CLP(\mathcal{FD})$. In *11th International Workshop on Functional and (Constraint) Logic Programming*. ENTCS vol. 76, 2002.

8. F. Fogolari, G. Esposito, P. Viglino, and S. Cattarinussi. Modeling of polypeptide chains as C-α chains, C-α chains with C-β, and C-α chains with ellipsoidal lateral chains. *Biophysical Journal*, 70:1183–1197 (1996).
9. K. Marriott and P. J. Stuckey. *Programming with Constraints.* The MIT Press, 1998.
10. S. Miyazawa and R. L. Jernigan. Residue-residue potentials with a favorable contact pair term and an unfavorable high packing density term, for simulation and threading. *Journal of Molecular Biology*, 256(3):623–644, 1996.
11. G. Raghunathan and R. L. Jernigan. Ideal architecture of residue packing and its observation in protein structures. *Protein Science*, 6:2072–2083, 1997.
12. B. Rost and C. Sander. Prediction of protein secondary structure at better than 70% accuracy. *Journal of Molecular Biology*, 232:584–599, 1993.
13. E. Shakhnovich L. Mirny. Protein folding theory: from lattice to all-atom models. *Annual Review of Biophysics and Biomolecular Structure*, 30:361–96, 2001.
14. J. Skolnick and A. Kolinski. Computational studies of protein folding. *Computing in Science and Engineering*, 3(5):40–50, 2001.
15. K. T. Simons, R. Bonneau, I. Ruczinski, and D. Baker. Ab initio protein structure prediction of CASP III targets using ROSETTA. *Proteins Suppl 3*, 171–176, 1999. `http://www.bioinfo.rpi.edu/~bystrc/hmmstr/server.php`
16. L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 2nd edition, 1997.
17. M. T. Swain and G. J. L. Kemp. A CLP approach to the protein side-chain placement problem. In T. Walsh, editor, *CP'01*, LNCS vol. 2239, pages 479–493, 2001.
18. Swedish Institute for Computer Science. Sicstus Prolog Home Page. `http://www.sics.se/sicstus/.`
19. T. Toma and S. Toma. Folding simulation of protein models on the structure-based cubo-octahedral lattice with the Contact Interactions algorithm. *Protein Science*, 8:196–202 (1999).
20. M. Vendruscolo, E. Kussell, and E. Domany. Recovery of Protein Structure from Contact Maps. *Folding and Design*, 2:295–306 (1997).
21. S. Will. Constraint-based Hydrophobic Core Construction for Protein Structure Prediction in the Face-Centered-Cubic Lattice Sebastian In R. B. Altman, A. K. Dunker, L. Hunter, T. E. Klein, eds., Proceedings of the *Pacific Symposium on Biocomputing*, 2002.

Gestures for Embodied Agents with Logic Programming

Zsòfia Ruttkay[1], Zhisheng Huang[2], and Anton Eliens[2]

[1] Center for Mathematics and Computer Sciences
Amsterdam, The Netherlands
Zsofia.Ruttkay@cwi.nl
http://www.cwi.nl/~zsofi

[2] Intelligent Multimedia Group, Division of Computer Science,
Vrije Universiteit Amsterdam, The Netherlands
{huang,eliens}@cs.vu.nl
http://www.cs.vu.nl/{~huang,~eliens}

Abstract. The paper discusses how distributed logic programming can be used to define and control the hand gestures of embodied agents in virtual worlds, by using the STEP language as an interface between the constructs of logic programming and the humanoid model defined in Virtual Reality Modelling Language (VRML). By using this framework, different gesture dictionaries can be defined and variants of a hand gesture, according to dynamically changing factors, can be generated on the fly. The approach is tested on the demanding demonstrator of conducting, providing experience, also on time performance of the animation.

1 Introduction

Embodied conversational agents (ECAs) [2] are more or less realistic representation of humans in interactive applications. Particularly, we are interested in Web-based applications, where the ECA is part of a virtual environment, accessible by standard web browsers. Such agents may occur in different applications and roles, as instructor or sales assistant, or representative of real persons in shared environments. For the latter case, the term avatar has been common in the literature. However, due to the novelty of the technology, this and other terms, such as humanoids and synthetic characters, are often used as synonyms. In course of the paper we use sometimes the term agent or avatar for brevity, instead of ECAs. It has been pointed out, that fully embodied conversational agents should have the capability of gesturing, both to increase efficiency and naturalness of communication [4]. Gestures are to express, redundantly or additively relative to what is being told, information about different aspects of the world [20, 29]:

- identify certain objects or events (e.g. the victory emblem);
- indicate characteristics of objects (location, shape, size), events (time, motion, start/end/repetition) or concepts (e.g. two hands with pointing fingers moving opposite to each other till touching, indicating ‘conflict’);
- show the state of the speaker (emotional, physical, cognitive, availability of modalities).

Gestures also improve the elicitation of spoken text, by punctuating it with special gestures to indicate syntactical chunks (contrast, enumeration, new topic) and communicational coordination (listening, turn giving/taking). Finally, gestures convey information about the identity of the speaker (cultural, gender, age, personality) [32].

The task of making an ECA to gesture is more complicated than defining and concatenating pre-cooked motion patterns. A single-hand gesture involves the coordinated motion of 18 hand and arm joints, and many variants of the same gesture occur. Moreover, for a single communicative function (e.g. greeting, pointing, emphasis) alternative gestures can be used, and the actual choice is made on the basis of static and dynamic characteristics of the speaker (e.g. handedness, is one of the hands full,...) and of the environment (e.g. relative location of speaker, listener and the object to be pointed at) and other factors of the presentation, such as the time is available to perform a gesture.

Summing up, the programming framework for the gesturing of agents should be appropriate, ideally, for all the following tasks:

1. provide high-level and compositional definition and control of gestures (as opposed to defining and controlling all joints individually);
2. allow the definition of gesture dictionaries, that is, all the gestures 'known' by an embodied agent;
3. allow to reason about the choice of hand and other parameters of the gesture to be used;
4. support the generation of gestures which are:
   a) individual,
   b) non-repetitive,
   c) resemble the motion qualities of humans [5],
   d) subtly synchronized (also to speech);
5. make possible the real-time generation of hand gestures in virtual environments.

We have a preference, in accordance with items 1-3, for a rule-based declarative language. Parallelism, obviously, is a must! Requirement 4 makes it clear that handling time and other numerical parameters is a necessity. The final requirement makes one ponder seriously if at all a logic-based declarative approach could be appropriate, because of the performance demands. As the ECA should be accessible by standard web browsers, it has to be modelled and, on the lowest level, animated in the Virtual Reality Modelling Language (VRML), a language notoriously known of tedious manipulation and low efficiency of rendering (but without better alternative).

In our paper we will outline how the STEP language, as an interface between distributed logic programming and VRML, can be used to deal with the above problems related to gesturing. (Note that our STEP language is different from the Standard for the Exchange of Product Model Data by ISO.) In Section 2 declarative programming and the DLP language is outlined, the STEP language is introduced and the modelling of humans in VRML is discussed. Section 3 is devoted to the definition and generation of gestures by using STEP. In Section 4 we sum up our approach and compare it to other methods for gesturing and discuss further possibilities. In particular, we report on runtime performance of demonstrator applications and the lessons learnt concerning synchronization. In the paper we use conducting as the running example, to demonstrate and experiment with the solutions given to the general problems of gesturing. This and some other examples of gesturing can be tried out on our web-page[1].

2 The Languages and Tools: DLP, Humanoids in VRML, and STEP

2.1 DLP

The Distributed Logic Programming Language (DLP) [6],[2] combines logic programming, object-oriented programming and parallelism. DLP has been used as a tool for web agents, in particular, 3D web agents [14]. The use of DLP as a language for the implementation of agent-based virtual environments is motivated by the following language characteristics: object-oriented Prolog, VRML External Authoring Interface extensions, and distribution. DLP is a high-level declarative language suitable for the construction of distributed software architectures in the domain of artificial intelligence, and in particular, for rule-based knowledge representation [7].

DLP is an extension of Prolog. For clauses and goals it uses standard Prolog (Edinburgh) syntax. In addition it provides a number of built-in primitives to support objects that may have so-called non-logical instance variables, to record the state of an object. There are explicit primitives to set or get the values of these non-logical instance variables. Also, the semantics of familiar operators, like the binary unification operator and the arithmetic assignment operator, are extended to allow them to handle non-logical instance variables, as well as plain Prolog logical variables and terms. In fact, substitution of the actual value of non-logical variables occurs automatically in any goal.

Objects are defined by their non-logical variables and a collection of clauses that act as the methods of an object. A method call is simply the invocation of a goal, which is resolved using the clauses defined for the object. Encapsulation of the state of the object, as recorded in the non-logical instance variables, is enforced by allowing only clauses defined within the object to access the non-logical instance variables. DLP supports multiple inheritance, that is objects may inherit collections of clauses from multiple objects that are defined as their ancestor(s). An object is designed as a set of rules and facts, which consist of a list of formulas built from predicates and terms (variables or constants). Objects in DLP are multithreaded. Each object has a thread that executes the own activity of the object, defined in the constructor for the object. In addition for each method call a new thread is started. The creation of a new thread is needed to allow for independent backtracking when multiple methods are activated. Mutual exclusion, that is protection against other method calls, is guaranteed until the first answer has been delivered to the caller. This limited form of protec-

[1] http://www.cwi.nl/~zsofi/gestures/

[2] http://www.cs.vu.nl/~eliens/projects/logic/index.html

tion requires that any updates of non-logical instance variables must take place while producing the first answer.

DLP supports communication by rendez-vous. The acceptance of a method call is determined by a synchronization or accept statement in the (active) body of the object, as defined in the constructor of the object. The accept statement allows for the unification of argument parameters, as well as conditions involving the state of the object. DLP allows for truly distributed objects, that is objects residing on different nodes in a computer network. As the only language of its kind, it supports distributed backtracking, the generation of (multiple) answers in a sequential fashion when backtracking occurs on the caller's side.

DLP is an extensible language. Special-purpose requirements for particular application domains can easily be integrated in the existing object-oriented language framework. DLP has been extended with a run-time library for VRML EAI (External Application Interface) and a TCP/IP library for the network communication.

The following predicates are some examples of DLP VRML built-in predicates:

- URL-load predicate *loadURL(URL)*
  loads a VRML world at URL into the Web browser.
- Get-position predicate *getPosition(Object,X,Y,Z)*
  gets the current position <*X,Y,Z*> of the Object in the VRML world.
- Set-position predicate *setPosition(Object,X,Y,Z)*
  sets the position <*X,Y,Z*> of the Object in the VRML world.
- Get-rotation predicate *getRotation(Object,X,Y,Z,R)*
  gets the current rotation <*X,Y,Z,R*> of the Object in the VRML world.
- Set-rotation predicate *setRotation(Object,X,Y,Z,R)*
  sets the rotation <*X,Y,Z,R*> of the Object in the VRML world.
- Get-property predicate *getSFVec3f(Object,Field,X,YZ)*
  gets a value (which consists of three float numbers *X, Y,* and Z) of the Field of the Object.
- Set-property predicate *setSFVec3f(Object,Field X,Y,Z)*
  assigns the SFVec3f value *X, Y*, and *Z* to the Field of the Object.

DLP programs are compiled to Java class files, which makes it a convenient tool for the implementation of VRML EAI[19] applets. DLP is also a distributed programming language. DLP programs can be executed at different computers in a distributed architecture by the support of its network communication library.

2.2 Humanoids in VRML

The avatars, as well as all items in the virtual world of 3D web applications are built in the Virtual Reality Modelling Language (VRML) [18] or X3D, the next generation of VRML. VRML is used to define the form, appearance and to some extent, motion of objects in the virtual world. Particularly, avatars have a humanoid appearance. The humanoid animation working group[3] proposes a specification, called H-anim specifi-

[3] http://h-anim.org

cation, for the creation of libraries of reusable humanoids in Web-based applications as well as authoring tools that make it easy to create humanoids and animate them in various ways.

According to the H-anim standard, an H-anim specification contains a set of *Joint nodes* that are arranged to form a hierarchy. Each Joint node can contain other Joint nodes and may also contain a *Segment node* which describes the geometry of the body part associated with that joint. Turning body parts of humanoids implies the setting of the corresponding joint's rotation. Moving the body part means the setting of the corresponding joint to a new position. H-anim specifies a standard way of representing humanoids in VRML, with different geometry and/or level of detail (LOD) of moving joints, ranging from 3D 'stick-figures' with only the main joints of the limbs moving (LOD 1), till realistic bodies, with all joints articulated for each finger (LOD 2).

The scripting language STEP has been designed for embodied agents/web agents which are based on H-anim humanoids.

2.3 The STEP Language

STEP (Scripting Technology for Embodied Persona) is a scripting language for H-anim web agents, in particular for their non-verbal acts like gestures and postures[15]. Based on dynamic logic [11], STEP has a solid semantic foundation, in spite of a rich number of variants of the compositional operators and interaction facilities on worlds.

The design of the scripting language STEP was motivated by the following principles listed below:

- convenience – may be used for non-professional authors;
- compositional semantics – combining operations;
- re-definability – for high-level specification of actions;
- parametrization – for the adaptation of actions;
- interaction – with a (virtual) environment.

The principle of convenience implies that STEP uses some natural-language-like terms for 3D graphics references. The principle of compositional semantics states that STEP has a set of built-in action operators. The principle of re-definability suggests that STEP should incorporate a rule-based specification system. The principle of parametrization justifies that STEP introduces a Prolog-like syntax. The principle of interaction requires that STEP is based on a more powerful meta-language, like DLP.

Turn and move are the two main primitive actions for body movements in STEP. Turn actions specify the change of the rotations of the body parts or the whole body over time, whereas move actions specify the change of the positions of the body parts or the whole body over time. Turn actions and move actions are expressed as follows:

turn(Agent,BodyPart,Direction,Duration)
move(Agent,BodyPart,Position,Duration)

In the above expressions *BodyPart* refers to a body joint in the H-anim specification, like l_shoulder, r_elbow, etc., *Direction* states a rotation, which can be an item with a form like rotation(1,0,0,-1.57), or 'front', a natural-language like term. *Position* states a position, which can be an item like position(1,0,0), or 'front', a natural-language like term. *Duration* states the time interval for the action, which can be a direct time specification, like time(2,second) and beat(2), or a natural-language like term, like 'fast', or 'slow'.

For instance, an action such as `turn(humanoid,l_shoulder,front, fast)` indicates turning the humanoid's left arm to the direction of front fast. The meaning of the natural language-like terms as such 'front' and 'fast' are defined by the ontology component in STEP. The STEP animation engine uses the standard slerp interpolation [33] (i.e., the spherical interpolation) to create rotations between a starting and ending orientation, by following the shortest path. The rotation interpolations are considered to be linear by default. STEP also supports non-linear interpolation by using the enumerating type of the interpolation operator. An example:

```
turnEx(Agent,l_shoulder,front,fast,enum([0,0.1,0.2,0.7,1])
```

The above expression turns the agent's left arm to the front via the interpolation points 0, 0.1, 0.2, 0.7, 1, that is, by covering equal parts of the entire trajectory in the indicated portions of the total time.

Scripting actions can be composed by using following composite operators:

- The sequence operator 'seq': the action *seq([*$Action_1$*, ...,*$Action_n$*])* denotes a composite action in which $Action_1$, ...,and $Action_n$ are executed sequentially.
- The parallel operator 'par': the action *par([*$Action_1$*, ...,*$Action_n$*])* denotes a composite action in which $Action_1$, ...,and $Action_n$ are executed simultaneously.
- The non-deterministic choice operator 'choice': the *action choice([*$Action_1$*, ...,* $Action_n$*])* denotes a composite action in which one of the $Action_1$, ..., $Action_n$ is selected and executed.
- Repeat operator 'repeat': the action *repeat(Action, N)* denotes a composite action in which the *Action* is repeated *N* times.

When using high-level interaction operators, scripting actions can directly interact with internal states of embodied agents or with external states of worlds. These interaction operators are based on a meta-language which is used to build embodied agents. Motivated from dynamic logic, the higher-level interaction operators in STEP are:

- Test-operator *test(p)*, which tests if *p* is true in a state,
- Do-operator *do(p),* which executes a goal *p* in the meta language level.
- Conditional *if_then_else(p,*$Action_1$*,*$Action_2$*),* which executes the $Action_1$ if the state *p* holds, otherwise executes the $Action_2$.

STEP has been implemented in the Distributed Logic Programming language DLP [8,13,14]. See [13] for the details of the implementation of STEP. XSTEP is the XML-encoded STEP language, which serves as a XML-based markup language for embodied agents [16]. The resources of STEP and XSTEP can be found on the STEP website[4].

[4] http://step.intelligent-multimedia.net

3 Definition of Hand Gestures

3.1 The Body Space and Reference Points

Human hand gesturing happens in certain parts of the space in front of and next to the body. In order to be able to express easily where a gesture should or may start/end, we use reference parameters and planes, relative to the human body, see Figure 1.

The body region planes, which are similar to the ones used in the HamNoSys sign language system [28], divide the space around the body into unit reference body regions. These, by label identified, regions will be used as possible start and end position of gestures. The semantics of the particular body space regions can be easily given in terms of the boundary spaces, see the example below and Figure 1.

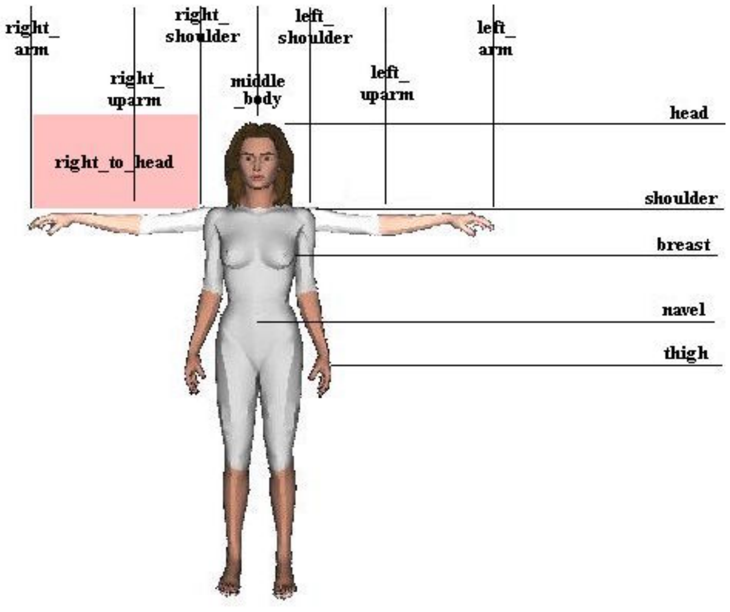


Fig. 1. Humanoid with body region planes and the body region right_to_head.

```
body_region(Agent,right_to_head,(X,Y,Z)):-
    above(Agent,shoulder,(X,Y,Z)),
    below(Agent,head,(X,Y,Z)),
    right_to(Agent,right_shoulder,(X,Y,Y)),
    left_to(Agent,right_arm,(X,Y,Z)).
below(Agent,BodyPlane,(_,Y,_)):-
    height(Agent,BodyPlane,PY),
    Y<PY.
```

A gesture usually may start at any position within a body region. This feature is useful when one wants to concatenate gestures, or generate variants by changing start and end positions. However, to avoid recomputation of start/end positions at each time, it is handy to have some default positions in each region. These can be either the middle of the region, or given explicitly for each region. Moreover, additional reference points on the body (point of features like of the mouth, nose, a point on the forehead) are handy to define self touching gestures like rubbing nose, thinking with forehead supported by one hand. These points are referred to by names, and given in terms of the geometry of the humanoid in question, but as a general, model-independent reference for hand gestures.

3.2 Stages of Hand Gesture Phrases

According to [24], a simple hand gesture is a 'phrase', consisting of the following 5 components, performed sequentially in time:

1. *Preparation:* the arm and hand are brought to the start position and configuration of the gesture.
2. *Pre-stroke hold:* the hand is waiting before the real characteristic motion is to be performed.
3. *Stroke:* the motion typical for the gesture is performed, between a start and an end position of the stroke, following a more or less restricted 3D trajectory and acceleration profile for the wrist, the orientation of the hand and the form of the handshape.
4. *Post-stroke hold:* the hand stands still in the end position for some time;
5. *Retraction:* the hand returns to the normal resting position.

Only the stroke is a compulsory and characteristic part of gestures. The preparation and retraction are treated as moving the hand between the current and a target location (start of the next gesture, or resting position), meanwhile changing possibly also hand shape. Special case is when a stroke is repeated (e.g. repeated beats). Then the preparation for the next stroke is the inverse of the last stroke performed.

In this section we first explain how a stroke is defined, then show how composite gestures can be built up from unit stokes and gestures. Finally we discuss the on the fly generation of gestures, that is the specification of gesture parameters responsible for fine-tuning and concatenating gestures.

3.2.1 Strokes

We define a stroke by the following characteristics:

- start and end *hand configuration*, that is, the value of rotation for all finger and wrist joints;
- *default start and end position for the wrist* (given explicitly or as a reference point);
- *possible start position for the wrist,* in terms of the body space region;
- *characteristics of the motion* transforming the hand and arm from the start to the end configuration, particularly: amplitude, dynamics, minimal, maximal and default duration; and the hand performing the gesture.

Note that an important characteristic, namely the trajectory of the wrist, is missing from the list of defining parameters. This is due to the fact that the trajectory is computed by STEP, by using the standard slerp interpolation between the start and end configurations for each joint. The rotation of the arm joints corresponding to a given position of the wrist are computed by STEP, using a fast analytical inverse kinematics method [15]. This method also takes care of setting the free degree of freedom for the human arm in a natural way.

For some gestures, the hand shape does not change during the stroke (pointing, beating with closed hands or pointing hands), for others the change of hand shape is characterized by the hand-shape at start and end (beat starting with pointing hand, ends with closed hand).The palm orientation is either constant, or is a transition from the start orientation to the end orientation. Thus motions made during the stroke may vary in complexity. Moreover, strokes of most of the gestures may be performed by any of the single hands (though humans have a preference for using usually the right hand), but for some gestures a specific hand or both hands need to be used (e.g. rubbing hands, showing conflict).

The hand configuration is given in terms of *hand shape* and *wrist orientation.* For specifying the *hand shape,* a pre-defined choice is offered, as a set of the common hand shapes required by the application. In our case, these include handshapes like '*ball*', '*straight*', '*conductor*', '*point*', '*o*', '*o_tube*', see Figure 2. Each of these hand shapes is given by a par construct in STEP:

```
script(set_fingers(Agent,conductor,Speed,r),Action) :-
   Action = par([
            turn(Agent,r_thumb1,rotation(0,1.2,1,1.2),Speed),
            turn(Agent,r_thumb2, down,Speed),
            turn(Agent,r_thumb3, down,Speed),
            turn(Agent,r_index1,down,Speed),
            turn(Agent,r_index2,down,Speed),
            turn(Agent,r_index3,down,Speed),
            turn(Agent,r_middle1,rotation(0,0,1,1.1),Speed),
            turn(Agent,r_middle2,rotation(0,0,1,0.5),Speed),
            turn(Agent,r_middle3,rotation(0,0,1,0.6),Speed),
            turn(Agent,r_ring1,rotation(0,0,1,1),Speed),
            turn(Agent,r_ring2,rotation(0,0,1,0.5),Speed),
            turn(Agent,r_ring3,down,Speed),
            turn(Agent,r_pinky1,rotation(0,0,1,0.9),Speed),
            turn(Agent,r_pinky2,rotation(0,0,1,0.5),Speed),
            turn(Agent,r_pinky3,down,Speed)])
            ,!.
```

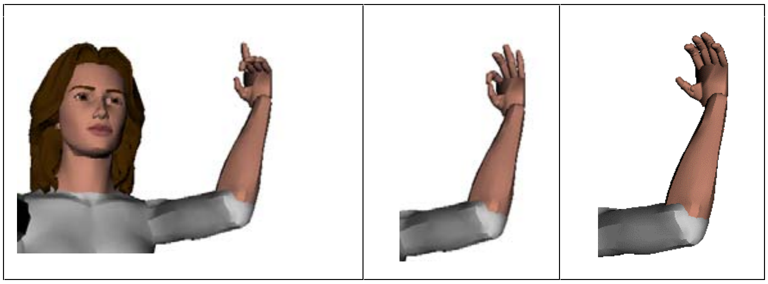


Fig. 2. Some handshapes used for conducting: *conductor*, *o* and *ball.*

The *wrist orientation* is given in terms of directions relative to the agent's body, like '*front*', '*front_up*', '*front_down*', '*in*', '*down*'. These orientations are associated by a particular wrist rotation, each.

The *default start and end locations* are points in the body space, while the *possible start location* is a body region or identical to the default start location. In the example below, the piano and the forte version of the dynamic stroke named 'beat' is defined. According to the definition, in the case of 'piano' intensity, any hand shape can be used and the start and end handshapes may differ, while in the 'forte' intensity version the hand shape should not change during the stroke, and should be one of the three given handshapes. Note that in this way a default handshape is defined too, namely if the clause is called with a free `HandShape` variable (see code segment below), it will get bounded to the first element of the list. Moreover, in this case the wrist moves more, from the '*front_down*' starting orientation to the '*front_back*' ending orientation. The '*elbow_front'* and '*elbow_front_up'* are points in the body space, namely those reached by turning the elbow in the directions indicated by the name. Similarly, '*arm_l_poin't* is a point in the body space reached by rotating both elbow and shoulder joints.

```
script(dynamic_stroke(beat,p,_StartH,_EndH,front_down,down,
    elbow_front_up,elbow_front,_StartRegion,_Dynamism),
    Action):-
        Action = [],!.
script(dynamic_stroke(beat,f,HandShape,HandShape,front_down,
    back_down,arm_l_front,elbowFront,_StartRegion,_Dynamism),
    Action):-
        Action=seq([do(in_list(HandShape,[conductor,point,fist])),!.
```

As said before, the motion trajectory is computed automatically by STEP. What is to be defined are the timing and other characteristics of the motion. Timing is to be given at each time a gesture is to be performed. The *duration* is to be given, when a gesture has to be performed, in terms of the default 1 second beat units of STEP, prescribing the usual and extreme durations of the stroke. For the *dynamism* of the gesture, we assume that for some time the motion is accelerating, then for some time a steady speed is kept, then decelerating to reach 0 velocity again for each joint. These stages are defined by specifying the time of the start and end point of the steady stage, in terms of percentage of the entire duration. The proportion of the covered part of the entire trajectory can be computed, and the *interpolation* function of STEP can be defined accordingly to achieve the required dynamical effect. Though any dynamical scheme can be defined, it is handy and sufficient to use a few predefined possibilities, such as: '*keep_start'* indicating a longer acceleration stage, '*keep_end'* indicating slow deceleration stage, and '*symmetrical*' indicating that acceleration and deceleration take place in 0.25 portion of the entire duration, each. By specifying long acceleration or deceleration times, the easy-in/ease-out effects, widely used in traditional animation, can be achieved. A specific case is the '*constant_speed'* case (resulting in non-human motion).

The *amplitude* of the gesture is to indicate the 'amount of arm motion'. We allow 3 qualitative categories, *p* (piano, minor arm motion), *m* (mezzo, normal arm motion) and *f* (forte, extra arm motion).

When a stroke has to be performed, a 'standard' stroke of a kind can be modified by the following parameters: duration, start location, start and end hand shape (if a choice was allowed), dynamism (if a choice was allowed), and precision. The first two parameters are to overwrite the corresponding default values. The third one is to specify start and end hand shape for gestures (for those which may be performed by several hand shapes). Dynamism may be set too. The *precision* numerical parameter is used to modify the start and end location of the arm joints and the dynamics. The standard Perlin noise [27] is used to modify the start and end location of the arm joints and the dynamics.

The timing of gestures is critical. Note that no variable is used to set the start time of a gesture. This is because it is assumed that gestures are performed in the order specified. If a gesture is to be started at e.g. 2 seconds on an absolute time line, then a *wait(2)* 'gesture' has to precede it, indicating that for 2 seconds nothing happens. We assume that the precise timing of the gestures – with respect to speech - is specified by a module preprocessing the text to be spoken which is marked up with some high-level gesture tags like the ones in our own GESTYLE language [25], or timing is provided by some module monitoring events.

3.2.2 Definition of Composite Gestures

Composite two-handed gestures are defined as parallel performance of right- and left-handed gestures, or as the sequence of one-handed gestures. In the first case, the stages of the two-handed gestures are declared in a par constructs. In the second case, the components are concatenated in such a way, that the retraction phase of one gesture is identical to the preparation of the following one. The repetitive stroke is a special case, when the end of the stroke and the beginning of the next one are connected by inverse strokes. The *inverse of a stroke* is defined by swapping the start and end configurations of the original stroke. In this way complex motion trajectories can be defined.

The parallel declaration of the motion of two hands induces synchronization, which can be further refined by introducing sequences of waits for specific durations. As durations can be expressions of variables involving also other durations, the timing of the motion of the two hands, and individual joints of it, can be declared in a fine granularity.

In the example below the main constructs for making strokes are given. The characteristics of strokes are given by the stroke(…) atomic predicates. The default dynamism is used from the stroke's definition, but the duration and hand are to be given.

```
script(perform_stroke(Agent,HandShape,WristOr,WristLoc,
    Dynamism,Duration,Hand),Action):-
  Action=par([
    set_arm(Agent,WristLoc,Dynamism,Duration,Hand),
    set_wrist(Agent,WristOr,Dynamism,Duration,Hand),
    set_fingers(Agent,HandShape,Dynamism,Duration,
      Hand)]))
]),!.
```

```
script(do_stroke(Agent,Stroke,Ampl,Sh,Duration,
    Hand),Action):-
  Action=seq([
    dynamic_stroke(Stroke,Ampl,Sh,Sw,Sa,Eh,Ew,Ea,_R,
      DefDyn),
    perform_stroke(Agent,Eh,Ew,Ea,DefDyn,Duration,
      Hand),
]),!.

script(do_closed_stroke(Agent,Stroke,Ampl,Sh,T0,
    T1,T2,Hand),Action):-
  Action=seq([
    dynamic_stroke(Stroke,Ampl,Sh,Sw,Sa,Eh,Ew,Ea,_R,
      DefDyn),
    perform_stroke(Agent,Eh,Ew,Ea,DefDyn,T1,Hand),
    wait(T0),
    inverse_dynamism(DefDyn, InvDyn),
    perform_stroke(Agent,Sh,Sw,Sa,Ampl,InvDyn,T2,Hand)
]),!.
```

3.3 Controlling the Gesturing

As introduced in the above paragraphs, we assume that whenever the embodied agent needs to do a gesture, the name of the gesture and information on timing are given. These instructions may have been prescribed by a user interactively or by an in advance prepared a script, or by some module responsible for controlling the behaviour of the agent. Further parameters necessary to instantiate a gesture may be given too, but not required.

3.3.1 Factors Effecting the Gesturing

The parameters missing to specify a gesture to be performed are decided by the *gesture control module*, on the basis of information present in the following three modules:

- The *gesture definition* module contains the definition of hand gestures, as explained in Section 3.2.
- The *avatar characteristics* module contains all static information about the avatar relevant for his gesturing, such as geometry of the body, handedness, specific motion characteristics (if any).
- The *context* module stores all dynamical information which is relevant for gesturing, both concerning objects of the world (e.g. location of an object to be pointed at), the avatar itself (e.g. if a hand is being occupied), and may be even information on resources available (e.g. processing capacity provided by the operating system).

The control module fills in parameters by using the default values given in the gesture definition, the default values associated with the character (e.g. default hand used to gesture, default motion profile characteristic for the gesturing of the character), and maybe according to dynamically changing characteristics of the context (e.g. the amplitude of conducting gesturing should correspond to the loudness of the music to be conducted). The value of a parameter can be derived by reasoning too (e.g. use the hand to conduct a musical phrase depending on the spatial location of instruments in an orchestra, unless for loud passages when both hands should be used). Note that the avatar characteristics module may contain only minimal information on the geometry of the avatar, while context module may be empty or missing entirely. Accordingly, the control module will make the gestures performed in a standard way, and indifferent of the world or performance context.

The final gesturing is thus produced by a reasoning module, using the definition of gestures, static defaults associated with a character (its body and its gesturing manners) and dynamical characteristics of the situation. This separation of ingredients, influencing the final gesturing has the advantage that a sequence of gesturing may be tuned to a new agent's body or gesturing habits. Also the gesture repertoire can be locally modified and extended.

3.3.2 Mechanisms for Controlling the Gesturing

The content of the control module should reflect the gesturing strategy of the application in question. Different gesturing strategies should be used for application domains like conduction, telling a story or explaining how to use a technical device. It is the responsibility of the author of the application to provide the 'right' domain rules for gesturing.

There are other standard services of the control module than gathering and interpreting information to fill in parameters. These are provided, and are in principle to be used for all applications. However, there may be cases when some features need to be switched off.

To avoid repetition of identical gestures, some parameters, like the hand shape to be used for gesturing, may be selected randomly from a limited choice, from time to time.

Another service of the gesturing control module is to 'fill the gaps' between gestures. Depending on the lengths of gaps between two prescribed strokes, either transitional motion is generated or, if the time is too long between the gestures, the hand is moved to a neutral position and from there via a preparatory motion to the start position of the next stroke.

It is possible to use gestures with reference to the objects in the environment. For instance, in order to perform a pointing gesture, the direction of pointing should be the direction of some, may be moving, object in the environment. By continuous update of the location of the object, the pointing hand can follow its motion. Note, however, that for the proper handling of a dynamical parameter like this, quite some reasoning and calculation may be needed. For instance, one has to decide if the pointing should be done at the hand only, or with also lower or full arm, and rotations for joints need to be recalculated accordingly. More on pointing gestures with STEP can be found in [17].

A special issue is handling overconstrained situations. Conflicts may involve different prescriptions for some part of the hand (e.g. the right hand is occupied with hold-ing an object, when a beat has to be performed), impossible short timing (with respect to speed characteristics of motion of joints, or performance limitations of the given rendering system), or some world obstacles on the motion trajectory. The handling of constraints can be done on the basis of some general, but possibly dynamical strategy to resolve them. The simplest strategy is: do perform the latest gesture. The phrasing of a gesture into preparation, stroke and retraction stages, and the definition of possible alternatives for location and shape of the hands at the beginning and end of the stroke ensure smoothness of the gesturing. For many applications this is sufficient.

More informed strategies to resolve conflicts can be declared in terms of importance of the different gestures, the alternatives for gestures in terms of hand usage (e.g., a right-handed person prefers the right hand to be used for beats, but if that hand is already in use, beat can be performed by left-hand too) or timing (e.g., the amplitude prescription may be scarified to meet prescribed timing, that is, instead of the intense big beat intended, a smaller one will be performed). The bottleneck in handling conflicts is not in the power of the language, but in the lack of descriptive knowledge of complex strategies and the context of gesturing, as done by humans.

4 Discussion

4.1 Related Work

The first embodied agents capable of hand gesturing were endowed with a fixed set of hand gestures [3, 23]. Badler and colleagues have been working on the parameterization of hand gestures, both concerning functionality [1] and motion characteristics [5]. Reasoning on gestures has been done in limited applications like weather presentation [25], or to choose the proper pointing gesture [22, 31]. For these applications, some own or commercial modeling and animation tool was used to render the agent. The work in [9], though dealing with fish world, set the tone for deriving behavior and motion from the perceived environment and the goals of the agent.

Recently, the extensible and modular definition of hand gestures gained attention [12, 21].

As of the high-level control of VRML agents, in [30] Constraint Programming was used to define and enforce internal and external conditions. The Signing avatars[5] is a commercial application to generate sign language on the Web.

Finally, one may be interested how our approach relates to the techniques used by professional computer animation filmmakers. They use high-end commercial modelling and animation tools like Maya[6], often extended with in-house developments for specific animation effects. These tools use (absolute) time-line as main organizing concept for the animation; pieces of motions, for different parts of the body can be inserted, with some smoothing, for different time periods. Scripting is used to define a sequence of predefined motions. Unlike in our case, neither automatic fine-tuning, nor

[5] http://www.vcom3d.com/

[6] http://www.alias.com/

on-demand behaviour and interactivity are supported. While the objectives of professional film makers, when aiming at a one-time high-quality animation film, are different than the ones stemming from Web-based applications, the power of the logic-based, declarative approach could be used to solve specific problems in computer animation films. Our approach could be used for generating the animation of interactive computer games.

4.2 Pros of Our Approach

We have shown, that the STEP language allows the declaration of gestures in a modular way, at different levels of detail. The decisions of what gesture to use and how to tune its parameters, can be drawn in dynamically changing environments. As STEP incorporates a high-level interface to the Humanoid definition in VRML, gesturing can be realized in VRML, by Humanoid characters, with different geometry and level of detail. This allows for designing transportable, standard embodied agents for Web applications. This very much wished for feature is addressed in a most recent summary on the state of the art [10], stating that in spite of the efforts of the Web3D consortium, embodied agents have not proliferated the Web, due to the lack of a easy to use programming interface to H-Anim.

4.3 On Parallelism, Synchronisation and Runtime Performance

Complex humanoid gestures are of a highly parallel nature. The STEP scripting language supports a direct way of modelling parallel gestures by offering a parallel construct (par), which results in the simultaneous execution of (possibly compound) actions.

To avoid unconstrained thread creation, the STEP engine makes use of a thread pool, containing a fixed number of threads, from which threads are allocated to actions. Once the action is finished, the thread is put back in the pool.

We have gained experience with several demonstrators running on a PC with a 550 mhz CPU and 128 MB memory, a low-end computer nowadays, under Windows NT running standard processes. We found that our approach works well for cases like football, tai chi demonstrator, and grasping, where the absolute time synchrony of the motions is not a critical issue. However, when many threads are needed for the coordination of the many joints involved in the motion, as in the conductor example (which requires approximately 60 threads), problems may occur concerning synchrony, in particular when there are many background jobs. When using parallelism three types of potential problems may be distinguished:

- synchronisation among gestures,
- (reliable) timing of gestures, and
- synchronisation of gestures with external events, e.g., audio.

Synchronisation among gestures requires that gestures that are meant to be executed simultaneously do indeed start and end at the same time. Unwanted delays however may occur, for example, when there are nested parallel constructs, due to the processing needed to unravel compound gestures. One remedy here is to optimize the Prolog engine, which is an ongoing effort. Another, equally obvious, remedy is to eliminate nested parallel constructs by some sort of flattening. These solutions require no or only a minor change to the STEP engine implementation. In effect, we have introduced a par_turn construct that allows declaring the parallel rotation of an arbitrary number of joints. However, such a construct, when applied exclusively, limits modelling to defining what can best be called successive 'frames' of a motion, which runs counter the approach advocated by STEP, namely defining gestures in a natural way, as concurrent motions of the human body. To solve the problem of reliable timing in a more fundamental way would require not only a modification of the STEP engine, but also a rather different implementation of the DLP threads supporting the parallelism in STEP. Currently, the implementation only allows for best effort parallelism and does not provide the means for deadline scheduling.

Synchronisation with external events is possible, to the extent that such external events are supported by the VRML/X3D environment. That does at this stage, unfortunately, not include the synchronisation with audio events, unless an audio track is broken up in separate pieces. One issue to work on, in the future, is the synchronisation with voice, using the events generated by a text-to-speech engine. For the time being, we have been using timing information generated by TTS engine to get gestures in sync with (long) speech audio. We achieved satisfactory performance on average PCs, and independent of the changes in resources provided by the operating system, by checking at the end of each gesture the actual time spent on it (which is often longer than intended), and adjusting the duration of following wait accordingly. For the conducting demonstrator this is not an improvement, as there is no wait between consecutive gestures.

Summarizing, as concerns the use of parallelism on modelling gestures we run into a number of problems which are either caused by exhausting the resources of the machine, inefficiencies in the execution of Prolog code, or limitations of the thread model supported by DLP. Partial remedies include, respectively, reducing the number of threads needed, eliminating nested parallel constructs, optimizing the Prolog execution engine, and a push-down of critical code of the STEP engine to the Java level. These measures can be taken without affecting the STEP engine or the underlying DLP thread model.

Nevertheless, there remains still some concern whether the inherent inefficiencies of the Java platform and the VRML Java External Authoring Interface can be overcome in a satisfying degree. Therefore, we have recently started to develop a version of DLP and STEP for the Microsoft .NET platform, which we believe that in combination with Direct3D based implementations of 3D graphic models allow for better control of the issues that we identified as problematic, in particular thread control, synchronization and interaction with a dynamic environment. As a preliminary result, we can mention that our Prolog implementation in the .NET C# language[7] already runs ten times faster that our Java implementation on a standard set of Prolog programs.

[7] http://www.microsoft.com/net

4.4 Future Work

In addition to improving the timing performance of STEP, there are several areas for improvement on the level of gesturing. The quality of the hand gesturing can be improved by implementing the usual constraints on human hand joints, and by taking care of collision avoidance (with other parts of the body). The more interesting further possibilities involve defining and experimenting with gesturing strategies, in order to decide when and what gesture is to be used. Thirdly, hand gestures should be used in coordination with other nonverbal modalities like head movements and facial expressions.
Finally, an avatar should be talking. Currently we are working on coupling the gesturing system in STEP with our GESTYLE markup language [26]. By using the latter, it is possible to indicate in a text to be spoken the gestures which should accompany the speech, or on yet a higher level, the meaning to be expressed by some gesture. Hence by end-users the gesturing system can be used from a high-level scripting of text, without any knowledge of STEP syntax.

Acknowledgement

Thanks to Cees Visser for his excellent technology support.

References

1. Badler, N., Bindiganavale, R., Allbeck, A., Schuler, W., Zhao, L., Palmer, M.: A Parameterized Action Representation for Virtual Human Agents. In: [2], (2000) 256–284
2. Cassell J., Sullivan J., Prevost S., Churchill E.: Embodied Conversational Agents. MIT Press, Cambridge MA. (2000)
3. Cassell, J., Bickmore, T., Billinghurst, M., Campbell, L., Chang, K., Vilhjálmsson, H., Yan, H.: Embodiment in Conversational Interfaces: Rea. Proceedings of ACM CHI 99. Pittsburgh, PA (1999) 520–527
4. Cassell, J., Thórisson, K.: The Power of a Nod and a Glance: Envelope vs. Emotional Feedback in Animated Conversational Agents. Applied AI 13(3) (1999) 519–538
5. Chi D., Costa M., Zhao L., Badler N.: The EMOTE Model for Effort and Shape. Proc. of Siggraph. New Orleans, Louisiana (2000) 173–182
6. Eliëns, A.: DLP, A Language for Distributed Logic Programming, Wiley, (1992)
7. Eliëns, A.: Principles of Object-Oriented Software Development. Addison-Wesley, (2000)
8. Eliëns, A., Huang, Z., Visser, C.: A Platform for Embodied Conversational Agents based on Distributed Logic Programming. Proceedings of AAMAS 2002 Workshop on Embodied conversational agents – let's specify and evaluate them! Montreal, Canada (2002)
9. Funge, J., Tu, X., Terzopoulos, D.: Cognitive Modeling: Knowledge, Reasoning, and Planning for Intelligent Characters. Proc. of SIGGRAPH '99, Los Angeles, CA. (1999) 29–38

10. Gratch, J., Rickel, J., Andre, J., Badler, N., Cassell, J., Petajan, E.: Creating Interactive Virtual Humans: Some Assembly Required. IEEE Intelligent Systems, July/August (2002) 54–63
11. Harel, D.: Dynamic Logic, Handbook of Philosophical Logic, Vol. II, D. Reidel Publishing Company, (1984) 497–604
12. Hartmann, B., Mancini, M., Pelachaud, C.: Formational Parameters and Adaptive Prototype Instantiation for MPEG-4 Compliant Gesture Synthesis. Proc. of Computer Animation 2002. Geneva, Switzerland (2002) 111–119
13. Huang, Z., Eliëns, A., Visser, C.: Implementation of a Scripting Language for VRML/X3D-based Embodied Agents. Proceedings of the 2003 Web3D Conference, Saint Malo, France. ACM Press, (2003) 91–100
14. Huang, Z., Eliëns, A., Visser, C.: Agent-based Virtual Communities. Proceedings of the 2002 Web3D Conference, Tempe, Arizona ACM Press, (2002) 137–144
15. Huang, Z., Eliëns, A., Visser, C.: STEP: a Scripting Language for Embodied Agents. In: Prendinger, H., Ishizuka, M. (eds.) Life-like Characters, Tools, Affective Functions and Applications, Springer-Verlag, Berlin Heidelberg New York (2003) 87–110
16. Huang, Z., Eliëns, A., Visser, C.: XSTEP: a Markup Language for Embodied Agents. Proceedings of the 16th International Conference on Computer Animation and Social Agents (CASA'2003), IEEE Press, (2003) 105–110
17. Huang, Z., Eliëns, A., Visser, C.: "Is it in my Reach?" – An Agents Perspective. In: Aylett, R., Ballin, D., Rist, T. (eds.), Proceedings of the 4th International Working Conference on Intelligent Virtual Agents (IVA'03), Lecture Notes in Artificial Intelligence Vol. 2792, Springer-Verlag, Berlin Heidelberg New York (2003) 150–158
18. ISO, VRML97: The Virtual Reality Modeling Language, ISO/IEC 14772-1/2, (1997)
19. ISO, VRML97: The Virtual Reality Modeling Language, Part 2: External Authoring Interface, ISO/IEC 14772-2, (1997)
20. Kendon, A.: Human gesture, In: Ingold, T., Gibson K. (eds.) Tools, Language and Intelligence. Cambridge University Press, Cambrige (1993)
21. Kopp, S., Wachsmuth, I.: Planning and Motion Control in Lifelike Gesture: a Refined Approach. Post-proceedings of Computer Animation 2000. Philadelphia, PA IEEE Computer Society Press (2000) 92–97
22. Lester, J., Voerman, J., Towns, S., Callaway, C.: Deictic Believability: Coordinated Gesture, Locomotion and Speech in Lifelike Pedagogical Agents. Applied AI 13(4/5) (1999) 383–414
23. Lundeberg, M., Beskow, J.: Developing a 3D-agent for the August Dialogue system. Proc. of AVSP'99, Santa Cruz, USA. (1999) 133–138
24. McNeill, D.: Hand and Mind: What Gestures Reveal about Thought. The University of Chicago Press, Chicago (1991)
25. Noma, T., Zhao, L., Badler, N.: Design of a Virtual Human Presenter. IEEE Computer Graphics and Applications 20(4) (2000) 79–85.
26. Noot, H., Ruttkay, Zs.: Style in Gesture. In: Camurri, A., Volpe, G. (eds.): Gesture-Based Communication in Human-Computer Interaction, Lecture Notes in Computer Science, Vol. 2915, Springer-Verlag, Berlin Heidelberg New York (to appear)
27. Perlin, K.: An Image Synthesizer. Computer Graphics 19(3) (1985) 287–296.
28. Prillwitz, S., Leven, R., Zienert, H.: Hamke, T. Henning, J.: HamNoSys Version 2.0: Hamburg Notation System for Sign Languages: An Introductory Guide. Volume 5, Signum Press, Hamburg (1989)
29. Poggi, I.: Mind Markers. In: Mueller C. and Posner R. (eds.): The Semantics and Pragmatics of Everyday Gestures. Berlin Verlag Arno Spitz, (2001)

30. Richard, N., Codognet, P.: The InViWo Virtual Agents. Proc. of Eurographics'99, Short Papers and Demos, Milan Italy, (1999)
31. Rickel, J., Johnson, L.: Animated Agents for Procedural Training in Virtual Reality: Perception, Cognition, and Motor Control. Applied Artificial Intelligence, 13(4/5) (1999) 343–382
32. Ruttkay Zs., Pelachaud, C., Poggi, I., Noot, H.: Exercises of Style for Virtual Humans. In: Canamero, L, Aylett, R. (eds.): Animating Expressive Characters for Social Interactions, John Benjamins Publishing Company, Amsterdam. to appear.
33. Shoemake K.: Animating Rotation with Quaternion Curves, Computer Graphics 19(3), (1985) 245–251

Author Index