Constraint Problem Specification as Compression

S. D. Prestwich[1], S. A. Tarim[2], and R. Rossi[3]

[1] Insight Centre for Data Analytics, University College Cork, Ireland
`s.prestwich@cs.ucc.ie`
[2] Department of Management, Cankaya University, Ankara, Turkey
`at@cankaya.edu.tr`
[3] University of Edinburgh Business School, Edinburgh, UK
`Roberto.Rossi@ed.ac.uk`

Abstract

Constraint Programming is a powerful and expressive framework for modelling and solving combinatorial problems. It is nevertheless not always easy to use, which has led to the development of high-level specification languages. We show that Constraint Logic Programming can be used as a meta-language to describe itself more compactly at a higher level of abstraction. This can produce problem descriptions of comparable size to those in existing specification languages, via techniques similar to those used in data compression. An advantage over existing specification languages is that, for a problem whose specification requires the solution of an auxiliary problem, a single specification can unify the two problems. Moreover, using a symbolic representation of domain values leads to a natural way of modelling channelling constraints.

1 Introduction

Constraint modeling is more of an art than a science, and considerable research has been devoted to making it easier for Constraint Programming (CP) users. A popular approach is to describe the problem in an abstract specification language, then transform the description into a concrete constraint model. Ideally a specification should be a concise but exact description of the problem, preferably in a formal language that is usually mathematical in nature. CP specification languages include OPL [21], Essence [11] and Zinc [14], while AMPL [10] is used to specify mathematical programs.

We propose a new approach to constraint problem specification: using Constraint Logic Programming (CLP) as a meta-language to describe CLP models. This approach has several advantages: (i) the user need know only one language (CLP); (ii) describing CLP variables and constraints as solutions to other Constraint Satisfaction Problems (CSPs) is very expressive, and avoids the need for separate software to generate complex specifications; (iii) domain values can be represented symbolically instead of numerically, making it easier to model channelling constraints; (iv) because CLP is a Turing-complete programming language further extensions are in principle unnecessary (though we find it useful to add a small number of features).

When writing specifications in this way, we can exploit any observed patterns in the model to make the specification more compact. Thus in our approach writing a short, clear specification requires the same kind of thinking as that used in data compression. G. Chaitin, one of the founders of Algorithmic Information Theory (AIT), argues that *a scientific or mathematical theory is a computer program for calculating the facts, and the smaller the program, the better*. This view has been summarised by the phrase *understanding is compression* [4]. In this paper we take the position that *specification is compression*. Like scientific theories, constraint problem specifications should be concise and easily understood. We call our specification language Kolmogorov because of this connection.

```
q3(V1,V2,V3) :-
   [V1,V2,V3]::1..3,
   V1#\=V2, V1#\=V3, V2#\=V3,
   V2-V1#\=1, V3-V1#\=2, V3-V2#\=1, V1-V2#\=1, V1-V3#\=2, V2-V3#\=1.
```

Figure 1: A CLP model for 3-queens

In Section 2 we introduce our approach using a trivial example. In Section 3 we demonstrate its usefulness on several examples from the CP literature. Section 4 discusses other approaches to problem specification. Section 5 concludes the paper.

2 Specifications as compressed CLP models

KOLMOGOROV uses CLP in two ways: as a (higher-level) specification language and as a (lower-level) constraint modeling language. In its high-level role it represents low-level concepts (variables, constraints and constants) as Prolog ground terms, and a few useful predicates are provided to aid description.

The specific CLP language we use is Eclipse [1] and we assume familiarity with basic CLP concepts. However, we now provide some notes for readers unfamiliar with Eclipse notation. `::` is used to declare variable domains, for example `[X,Y]::1..3` means that variables `X` and `Y` have finite domain $\{1, 2, 3\}$. Equalities, disequalities and inequalities between finite domain variables are expressed by operators such as `#=`, `#\=` and `#<`. Functions of lists such as `sum` and `max` can be used in constraints. A constraint expression is true or false, represented by values 1 and 0 respectively, and these values can be used in reified expressions such as `(X#=5)+(Y#<3)#<2`, which means that at most one of the constraints `(X#=5)` and `(Y#<3)` must be satisfied. A declaration `intset(V,A,B)` means that variable `V` has a set domain which is all the subsets of integers between `A` and `B`. For set variables `#` denotes cardinality and `/\` intersection. The predicate `indomain` nondeterministically assigns a domain value to its argument variable. The predicate `findall` collects solutions via backtracking, for example if variable `I` has domain `[1,2,3]` then `findall(f(I),(indomain(I),I#\=2),L)` instantiates `L` to the list `[f(1),f(3)]`.

As an introductory example we use the well-known N-queens problem, which uses a generalised chess board with a grid of $N \times N$ squares. The problem is to place N queens on it in such a way that no queen attacks any other. A queen *attacks* another if it is on the same row, column or diagonal (in which case both attack each other). A classic paper [16] presented 9 CSP models called Q1–Q9 and we use the popular Q1. For each row i of the board define a variable v_i with domain $\{1, \dots, N\}$. An assignment $v_i = j$ means that a queen is placed at row i column j. Because a variable can only take one value, this model already implies that no row can contain two queens. We need constraints to ensure that no two columns contain a queen: $v_i \neq v_j$ for $1 \leq i < j \leq N$. Similarly for diagonals: $v_i - v_j \neq i - j$ for $1 \leq i < j \neq N$. An Eclipse model for this problem with $N = 3$ is shown in Figure 1. This is probably the simplest and clearest model, though it is not general-purpose because the number of queens is fixed to 3.

Now consider a naive KOLMOGOROV specification of this model, shown in Figure 2. It describes the problem variables (via `kvar`), and the subgoals describing variable declarations and constraints (via `kgoal`), with the CLP variables `V1`, `V2` and `V3` replaced by structured ground terms `v(1)`, `v(2)` and `v(3)`. (We do not model the head `q3(V1,V2,V3)` of the CLP clause.) The `[]` argument is used for passing parameters such as the size of the chess board

```
kvar(v(1),[]).
kvar(v(2),[]).
kvar(v(3),[]).

kgoal([v(1),v(2),v(3)]::1..3,[]).
kgoal(v(1)#\=v(2),[]).      kgoal(v(1)#\=v(3),[]).     kgoal(v(2)#\=v(3),[]).
kgoal(v(2)-V(1)#\=1,[]).    kgoal(v(1)-V(2)#\=1,[]).   kgoal(v(3)-V(1)#\=2,[]).
kgoal(v(1)-V(3)#\=2,[]).    kgoal(v(3)-V(2)#\=1,[]).   kgoal(v(2)-V(3)#\=1,[]).
```

Figure 2: Naive KOLMOGOROV specification for 3-queens

and will be used below. The meaning of this model is very simple. The terms representing the variables of the 3-queens CLP model are the solutions of the goal

```
?- kvar(V,[]).
```

These solutions are:

```
V = v(1)
V = v(2)
V = v(3)
```

Similarly, the constraints and variable declarations of the model are the solutions of the goal

```
?- kgoal(C,[]).
```

which are:

```
C = [v(1),v(2),v(3)]::1..3
C = (v(1)#\=v(2))
...
C = (v(2)-v(3)#\=1)
```

To create a CLP model that can be passed to a solver, the `kgoal` solutions are collected into a list $\mathcal{L}$ via backtracking, and variable terms such as `v(2)` are replaced by CLP variable names such as `V2`. To do this, our compiler traverses the terms in $\mathcal{L}$ and replaces each `kvar` solution term by a CLP variable, using a hash table to keep track of the names. The result is exactly the model in Figure 1.

The above example illustrates the KOLMOGOROV framework, but so far we have not demonstrated any advantage because the specification is longer and less clear than the CLP model itself. We now compress the description by looking for patterns. Observe that each variable `v(I)` occurs in a `#\=` constraint with each variable `v(J)` where `I<J`, and that each variable occurs in another `#\=` constraint with each variable `v(J)` where I$\neq$J. We can exploit this simple pattern to produce the more compact specification in Figure 3. The `kvar` and `kgoal` solutions of this specification are the same as those of the previous one, and exactly the same CLP model will be generated.

Figure 3 uses a predicate `csp` which we provide as part of KOLMOGOROV. This predicate posts the constraints in its argument then nondeterministically solves the corresponding CSP by nondeterministically assigning values to its variables. Although any CLP code can be used in the body of a `kvar` or `kgoal` clause, often it will be a CSP so this predicate makes it easier to write specifications. We shall show that specifying variables and constraints as the solutions to `kvar` and `kgoal` CSP solutions makes KOLMOGOROV very expressive. We refer to these as *auxiliary CSPs*.

```
kvar(v(I),[]) :- csp(I::1..3).

kgoal(L::1..3,[]) :- findall(V,kvar(v(I),[]),L).
kgoal(v(I)#\=v(J),[]) :- kvar(v(I),3), kvar(v(J),3), I<J.
kgoal(v(J)-v(I)#\=D,[]) :- kvar(v(I),3), kvar(v(J),3), I\=J, D is J=I.
```

Figure 3: Compressed Kolmogorov specification for 3-queens

```
kvar(v(I),[N]) :- csp(I::1..N).

kgoal(L::1..N,[N]) :- findall(V,kvar(v(I),[N]),L).
kgoal(v(I)#\=v(J),[N]) :- kvar(v(I),N), kvar(v(J),N), I<J.
kgoal(v(J)-v(I)#\=D,[N]) :- kvar(v(I),N), kvar(v(J),N), I\=J, D is J-I.
```

Figure 4: Compressed Kolmogorov specification for N-queens

As a final step the compact 3-queens specification can be generalised to the N-queens problem, as shown in Figure 4 where the board size `N` is specified in the second argument of `kvar` and `kgoal`. The specification is now close to a mathematical description of the problem. This trivial example illustrates our approach: we detect and exploit patterns in the model in order to obtain a more compact representation, which is also more amenable to generalisation. However, in practice we need not start with a model and transform it, as in this example: familiarity with Kolmogorov means that we can write a compact specification directly.

Exploiting patterns to obtain a more compact representation is precisely what is done in data compression (though typically using different techniques). This is why we consider Kolmogorov specifications to be *compressed constraint models*. In most of our examples we shall exploit patterns among constraints by expressing them as solutions to auxiliary CSPs, which naturally capture most of the patterns in our problems. However, for some specifications we might require a more algorithmic style of compression, exploiting the fact that CLP is a programming language as well as a constraint solver. In principle *any* pattern in a constraint model can be exploited by a Kolmogorov specification, because CLP is a Turing complete language so it can express any form of algorithmic compression.

In this paper we shall gloss over some details that in practice also require handling: an objective function (if any), the search strategy, library declarations, and which variables form the part of the solution we are interested in (usually declared in a goal, such as `q3(V1,V2,V3)` in the model of Section 1). We shall focus on specifying constraint satisfaction problems.

3 Case studies

We now present Kolmogorov specifications for several problems from the CP literature, introducing additional features and pointing out its advantages.

3.1 Four standard problems

An Essence paper [11] presented specifications for four problems (the knapsack problem, Golomb rulers, SONET and the social golfer) and we start by modeling the same problems. As we have not modeled objective functions we consider them as decision problems, but it would be

```
kvar(u(I),[B,K,N,SL,VL]) :- csp(I::1..N).

kgoal((UL::0..1,sum(S1)#=<B,sum(S2)#>=K),[B,K,N,SL,VL]) :-
   findall(u(I),kvar(u(I),[B,K,N,SL,VL]),UL),
   findall(U*S,csp((element(Q,UL,U),element(Q,SL,S))),S1),
   findall(U*V,csp((element(Q,UL,U),element(Q,VL,V))),S2).
```

Figure 5: Kolmogorov specification of the knapsack problem.

```
kvar(t(I),[N,M]) :- csp(I::1..N).
kvar(d(I,J),[N,M]) :- kvar(t(I),[N,M]), kvar(t(J),[N,M]), I<J.

kgoal(t(I)::0..M,[N,M]) :- kvar(t(I),[N,M]).
kgoal(d(I,J)::1..M,[N,M]) :- kvar(d(I,J),[N,M]).
kgoal(d(I,J)#=t(J)-t(I),[N,M]) :- kvar(d(I,J),[N,M]).
kgoal(ordered(TL),[N,M]) :- findall(t(I),kvar(t(I),[N,M]),TL).
kgoal(alldifferent(DL),[N,M]) :- findall(d(I,J),kvar(d(I,J),[N,M]),DL).
```

Figure 6: Kolmogorov specification of the Golomb ruler problem.

simple to extend Kolmogorov to optimisation problems. We do not explain these problems as they are standard CP examples, and our aim here is simply to show that Kolmogorov can model them as easily as other specification languages (but see Section 3.2 for a description of the social golfer problem).

The Knapsack problem is specified in Figure 5. It has parameters `B` (knapsack capacity) and `K` (minimum total value), a list of item sizes `SL`, a list of item values `VL`, and a desired set cardinality `N`. The Golomb ruler problem is specified in Figure 6 using a CP model with auxiliary variables from [20], with `N` ticks and a ruler of length `M`. The SONET problem is specified in Figure 7. This is a decision version of the unlimited traffic capacity model in [18] with an upper bound `S` on the objective. The social golfer problem is specified in Figure 8 based on the model of [8, 12].

```
kvar(n(I),[N,M,S,R]) :- csp(I::1..N).
kvar(r(K),[N,M,S,R]) :- csp(K::1..M).
kvar(x(I,K),[N,M,S,R]) :- csp((I::1..N,K::1..M)).

kgoal(intset(n(I),1,M),[N,M,S,R]) :- kvar(n(I),[N,M,S,R]).
kgoal((intset(r(K),1,N),#(r(K),Q),Q#=<R),[N,M,S,R]) :- kvar(r(K),[N,M,S,R]).
kgoal(x(I,J)::0..1,[N,M,S,R]) :- kvar(x(I,J),[N,M,S,R]).
kgoal(sum(XL)#=<S,[N,M,S,R]) :- findall(x(I,J),kvar(x(I,J),[N,M,S,R]),XL).
kgoal(x(I,K)#=((I in r(K))*(K in n(I))),[N,M,S,R]) :- kvar(x(I,K),[N,M,S,R]).
kgoal((#(n(I) /\ n(J),Q),Q#>=1),[N,M,S,R]) :-
   kvar(n(I),[N,M,S,R]), kvar(n(J),[N,M,S,R]), I<J.
```

Figure 7: Kolmogorov specification of the SONET problem.

```
kvar(g(I,J),[W,G,S]) :- csp((I::1..W,J::1..G)).

kgoal(intset(g(I,J),1,P),[W,G,S]) :-
   kvar(g(I,J),[W,G,S]), csp((P#=G*S,#(g(I,J),S))).
kgoal((g(I,J) disjoint g(I,J1)),[W,G,S]) :-
   kvar(g(I,J),[W,G,S]), kvar(g(I,J1),[W,G,S]), J<J1.
kgoal((#(g(I,J) /\ g(I1,J1),N),N#=<1),[W,G,S]) :-
   kvar(g(I,J),[W,G,S]), kvar(g(I,J1),[W,G,S]), I<I1.
```

Figure 8: Kolmogorov specification of the social golfer problem.

```
kvar(g(I,J),[W,G,S]) :- csp((I::1..W,J::1..G)).

kconst(s(PL),[W,G,S]) :- length(PL,S), csp((PL::1..G*S,ordered(PL))).

kgoal((V::Dom),[W,G,S]) :-
   findall(C,kconst(C,[W,G,S]),Dom), kvar(V,[W,G,S]).
kgoal((g(I,J1)#=s(PL1))+(g(I,J2)#=s(PL2))#<2,[W,G,S]) :-
   csp((I::1..W,[J1,J2]::1..G,J1#<J2)), kconst(s(PL1),[W,G,S]),
   kconst(s(PL2),[W,G,S]), intersection(PL1,PL2,[_|_]).
kgoal((g(I1,J1)#=s(PL1))+(g(I2,J2)#=s(PL2))#<2,[W,G,S]) :-
   csp(([I1,I2]::1..W,[J1,J2]::1..G)), kconst(s(PL1),[W,G,S]),
   kconst(s(PL2),[W,G,S]), intersection(PL1,PL2,[_,_|_]).
```

Figure 9: Kolmogorov specification for the improved social golfer

3.2 Improved social golfer

The specification for the social golfer problem in Figure 8 is based on a standard model, but an interesting improved model was reported by Puget [17]. We shall use this model to illustrate two powerful features of Kolmogorov: *symbolic constants* to simplify the writing of constraints, and incorporating an *auxiliary CSP* (used to generate data for the main problem) into a specification.

The problem is as follows. In a golf club there are G groups each with S golfers who play every week, and we must find a schedule of W weeks such that no two of the $G \times S$ golfers meet more than once. Figure 9 shows a Kolmogorov specification for a version of Puget's model, using finite domain variables instead of set variables. For each week and group we define an integer variable whose domain represent all possible groups. We post binary constraints to ensure that the groups in a week do not intersect, and that groups from different weeks have at most one player in their intersection. We omit symmetry breaking constraints.

This example introduces a new Kolmogorov feature: a predicate `kconst` for representing constants (such as integers) symbolically by ground terms. Here `kconst` declares that any term of the form `s(PL)`, where `PL` is a list of length `S`, represents an integer; it does not matter what value this integer is, as long as each different term maps to a unique integer (these are automatically generated during Kolmogorov compilation). So symbolic constants such as `s([0,1,2])`, `s([0,1,3])`, `s([0,1,4])`,... represent groups, and are replaced by integers `1`, `2`, `3`,... during compilation, which form domains for the `g`-variables. The advantage of symbolic constants is that we can write some constraints in a very natural way: for example, to check whether integers (say 79 and 335) assigned to two `g`-variables represent intersecting sets,

```
kvar(c(P),[B,PL,DL]) :- csp(P::1..6).

kgoal(c(P)::1..B,[B,PL,DL]) :-
   kvar(c(P),[B,PL,DL]).
kgoal(sum(XPL)#>=D,[B,PL,DL]) :-
   csp((Q::1..3,element(Q,DL,D))),
   findall(X*P,(member(X,PL),csp(element(Q,PL,P))),XPL).
```

Figure 10: KOLMOGOROV specification for cutting stock

we simply check whether their symbolic constants (say `s([3,4,7])` and `s([2,4,8])`) contain elements in common (in this case they both contain `4`) as in Figure 9.

Another advantage of our approach is that the groups need not be generated in a preprocessing phase. Their generation is part of the KOLMOGOROV specification, and occurs automatically when the CSP in the `kconst` clause is solved during compilation. This feature can also be useful for industrial problems, as we show in Section 3.3.

3.3 Cutting stock problem

To further illustrate the usefulness of auxiliary CSPs, we use a well-known industrial problem: the cutting stock problem. This example is taken from H. Kjellerstrand's MiniZinc page.[1] A company cuts boards of size 17 into pieces of sizes 3, 5 and 9, and they must cut enough pieces to satisfy demands 25, 20 and 15 respectively. There are six feasible cutting patterns for a board:

size 3	5	4	2	2	1	0
size 5	0	1	2	0	1	3
size 9	0	0	0	1	1	0
wasteage	2	0	1	2	0	2

where wasteage is the material left after cutting pieces from the board. We must decide how many boards to cut, and how many times to apply each cutting pattern. We turn this into a decision problem by fixing the number of boards to `B`. A KOLMOGOROV specification is shown in Figure 10, and to generate a constraint model we call

```
?- kgoal(C,[B,PL,DL]).
```

with `B` set to some integer and

```
PL = [[5,0,0],[4,1,0],[2,2,0],[2,0,1],[1,1,1],[0,3,0]]
DL = [25,20,15]
```

The cutting patterns are provided here as a list parameter, and in the MiniZinc specification as a matrix.

In real-world instances the set of feasible cutting patterns is not random, but may be (for example) a consequence of the design of the cutting machinery, or the need to avoid excess wasteage. If we can model this machinery it might be possible to derive an auxiliary CSP whose solutions are the feasible cutting patterns. For example, suppose we wish to allow any cutting pattern `[U,V,W]` with wasteage less than 3. We can then make cutting pattern generation part of the specification by expressing it as a CSP and omit the `PL` parameter, as in Figure 11.

[1]http://www.hakank.org/minizinc

```
kvar(c(P),[B,DL]) :- csp(P::1..6).

kgoal(c(P)::1..B,[B,DL]) :-
   kvar(c(P),[B,DL]).
kgoal(sum(XPL)#>=D,[B,DL]) :-
   csp((Q::1..3,element(Q,DL,D))), findall(X*Y,csp(pattern(U,V,W)),XPL).

pattern(U,V,W) :-
   [U,V,W]::0..6, Used#=U*3+V*5+W*9, Waste#=17-Used, Used#=<17, Waste#<3.
```

Figure 11: Kolmogorov specification for cutting stock with implicit patterns

3.4 Covering arrays

Kolmogorov's symbolic constants are helpful when we have models involving *compound* or *dual variables* [19] and *channeling constraints* [5]. As an example we use a published CP model for covering arrays [13]. This is not the *naive* model which is the simplest to describe (see [11] for an Essence specification) but does not scale up to large instances, but the *hybrid* model designed for scalability, which was used to extend known results for covering arrays.

The problem is as follows. A covering array $CA(t, k, g)$ of size b is an $b \times k$ array consisting of b vectors of length k with entries from $\mathbb{Z}_g = \{0, 1, \ldots, g-1\}$ (g is the size of the alphabet) such that every one of the g^t possible vectors of size t occurs at least once in every possible selection of t elements from the vectors. The objective is to find the minimum b for which a $CA(t, k, g)$ of size k exists, and fixing b gives a decision problem.

The obvious way of modeling the problem uses a set of decision variables $x_{i,j} \in \{0, \ldots, g-1\}$ to represent the covering array. In [13] this is referred to as the *naive model* because it does not scale well to large instances, because the coverage constraints are hard to express efficiently. An *alternate* model instead uses a $\binom{k}{t} \times b$ matrix A of integers in $\mathbb{Z}_{g^t}$, which is represented by another set of Boolean variables: for each column c, row j and value y define a variable a_{cjy}. The idea of the a-variables is that a choice of t columns from the k columns in the covering array is represented by a single integer $c \in \mathbb{Z}_{\binom{k}{t}}$, and that the values in these t columns are combined to give a single integer $y \in \mathbb{Z}_{g^t}$. The a-variables model this alternative representation of the covering array. We call the a-variables *compound variables* and they occur in many constraint models. However, the alternate model is also inefficient because it requires a large number of *intersection* constraints to ensure consistency between a-variables that share x-variables.

The *hybrid* model combines both representations: the coverage constraints are expressed on the a-variables, and channeling constraints between the a- and x-variables make the intersection constraints redundant. Figure 12 shows a Kolmogorov specification for this model (apart from symmetry breaking constraints which we omit here). Instead of indexing the a-variables by an integer $c \in \mathbb{Z}_{\binom{k}{t}}$ to describe the choice of columns, we index them by a list of the columns: $a_{j,i_1,\ldots,i_t}$. The a-domains are integers, and a simple way of choosing these integers is to post intentional non-binary channeling constraints

$$a_{j,i_1,\ldots,i_t} = \sum_{i=0}^{t-1} 2^i x_{i,j}$$

as mentioned in [13]. However, an extensional method has the advantage of stronger filtering:

```
kvar(x(I,J),[T,K,G,B]) :- csp((I::1..K,J::1..B)).
kvar(a(J,IL),[T,K,G,B]) :- length(IL,T), csp((J::1..B,IL::1..K,ordered(IL))).

kconst(c(CL),[T,K,G,B]) :- length(CL,T), csp(CL::0..G-1).

kgoal(x(I,J)::0..G-1,[T,K,G,B]) :-
   kvar(x(I,J),[T,K,G,B]).
kgoal(a(J,IL)::Dom,[T,K,G,B]) :-
   kvar(a(J,IL),[T,K,G,B]), findall(c(CL),kconst(c(CL),[T,K,G,B]),Dom).
kgoal((a(J,IL)#=c(CL))+(x(I,J)#=A)#<2,[T,K,G,B]) :-
   kvar(a(J,IL),[T,K,G,B]), kconst(c(CL),[T,K,G,B]),
   csp((element(Q,IL,I),element(Q,CL,A))).
kgoal(gcc(BL,YL),[T,K,G,B]) :-
   findall(gcc(1,B,c(CL)),kconst(c(CL),[T,K,G,B]),BL),
   findall(a(J,IL),kvar(a(J,IL),[T,K,G,B]),YL).
```

Figure 12: Kolmogorov specification for hybrid CA model

post binary constraints to forbid nogoods

$$\langle a_{j,i_1,\ldots,i_t} = c,\ x_{i_q,j} = c' \rangle$$

where c is any value corresponding to the assignment $x_{i_q,j} = c'$. A difficulty here is that it is not trivial to write a mathematical relationship between c and c' (and this was not explicitly done in [13]). But the difficulty vanishes if we use `kconst` to define symbolic constants as in Section 3.2. We represent the elements of each a-domain by structured terms: $a_{j,i_1,\ldots,i_t} = c(c_1,\ldots,c_t)$ written as `c(CL)` where `CL` is a list of integers. Now the binary channeling constraints nogoods are simple to state:

$$\langle a_{j,i_1,\ldots,i_t} = c(c_1,\ldots,c_t),\ x_{i_q,j} = c_q \rangle$$

for $q = 1,\ldots,t$. Representing domain integers by symbolic constants allows us to specify channeling constraints in a more natural way, without the need for devising complicated relationships between domains.

An interesting generalisation of covering arrays is *Quilting arrays* [6] in which we do not need to cover all patterns, but only those with a specified pattern such as using only two values, or with all different values. We can easily extend the Kolmogorov specification of Figure 12 to handle Quilting arrays by adding a constraint such as `alldifferent(CL)` to the `kconst` definition. This prevents any compound variable a from taking a value that corresponds to an invalid pattern of x assignments.

4 Related work

CLP languages themselves were initially promoted as high-level specification languages, until the need for greater abstraction became apparent. Kolmogorov is perhaps closest in spirit to NP-Spec [3], which uses Datalog (a simplified form of Prolog without structured terms) plus some second-order predicates to specify problems. However, NP-Spec and Kolmogorov specification look very different, as can be seen by comparing the models for the social golfer problem in Figures 8 and 16. Answer Set Programming [2] and Business Rules [7] have also been used as specification languages for CP. A different approach is taken by Essence [11], Zinc [14], OPL [21], ESRA [9], $\mathcal{F}$ and LOCALIZER [15], which use mathematical language to obtain highly abstract specifications. AMPL [10] and other languages play a similar role for mathematical programming.

An important feature of some languages (ESSENCE, ESRA, $\mathcal{F}$ and LOCALIZER) is quantification over decision variables, rather than merely over ranges of integers [11]. KOLMOGOROV does not contain explicit quantifiers, but because it represents variables (also constraints and symbolic constants) as generic Prolog terms it has similar expressive power: it can enumerate all variables whose representation matches a given term. This occurs in our specifications when `kvar` is called from a `kgoal` clause.

We claim that KOLMOGOROV specifications are typically of comparable size to those in other specification languages. To support this claim, and to show that KOLMOGOROV is quite different to existing specification languages, Appendix A reproduces models from [11] for the social golfer problem in Zinc, ESRA, OPL, NP-SPEC and ESSENCE. There is no generally-agreed way of comparing the relative sizes of specifications in such different languages, but the KOLMOGOROV specification (Figure 8) is clearly of a typical size. Whether it is easier or harder to understand is difficult to establish, and depends partly on whether the reader has CLP experience, but we note that it uses a known syntax (CLP) without the need for special symbols or typefaces, or the mathematics of sets and functions.

5 Conclusion

KOLMOGOROV is a new approach to writing specifications for constraint problems. Instead of creating a new mathematical language we use CLP as a meta-language to describe CLP models at a higher level of abstraction. Thus a KOLMOGOROV specification is a CLP description of a CLP model, which exploits patterns in the model to make the description clear and compact.

It might be objected that KOLMOGOROV is not a specification language at all, as our specifications are written in an existing programming language. But we have shown that for a variety of problems its specifications are of comparable size to those of other specification languages. We argue that the purpose of a specification is to describe a problem clearly, precisely and succinctly, and that KOLMOGOROV fulfils these criteria. Our argument assumes familiarity with CLP, which is of course not true of all CP users. However, our approach requires the modeler to know only one language, whereas most approaches require knowledge of both a CP language and a very different specification language.

Our use of CLP to specify constraint problems could be criticised on the grounds that CLP contains non-declarative features (such as negation-as-failure, `findall` and the cut) whereas specification languages are typically declarative. We chose to use a full programming language for reasons of convenience and compactness, but we could have used only declarative features. Horn clause logic is a subset of CLP that is also Turing complete, and restricting KOLMOGOROV to this language would have the advantage of being completely declarative and with a very simple syntax.

6 Acknowledgments

This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289.

References

[1] K. R. Apt and M. Wallace. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, 2007.

[2] M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *ICLP'09 Workshop on Answer Set Programming and Other Computing Paradigms*, 2009.

[3] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-Spec: an executable specification language for solving all problems in NP. *Computer Languages*, 26(2-4):165–195, 2000.

[4] G. Chaitin. Epistemology as information theory: from Leibniz to Ω. *Collapse*, 1:27–51, 2006.

[5] B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2):167–192, 1999.

[6] C. J. Colbourn and J. Zhou. Improving two recursive constructions for covering arrays. *Journal of Statistical Theory and Practice*, 6:30–47, 2012.

[7] F. Fages and J. Martin. From rules to constraint programs with the rules2cp modelling language. In *Recent Advances in Constraints: 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming*, volume 5655 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2008.

[8] T. Fahle, S. Shamberger, and M. Sellmann. Symmetry breaking. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2001.

[9] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Proceedings of the 13th International Symposium on Logic Based Program Synthesis and Transformation*, pages 214–232, 2003.

[10] R. Fourer, D. Gay, and B. W. Kernighan. *AMPL: a Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.

[11] A. M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. ESSENCE: a constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

[12] W. Harvey. Symmetry breaking and the social golfer problem. In *CP'01 Workshop on Symmetries*, 2001.

[13] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(3):199–219, 2006.

[14] K. Marriott, N. Nethercote, R. Rafeh, P.J. Stuckey, M. Garcia de la Banda, and M. Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[15] L. Michel and P. van Hentenryck. Localizer. *Constraints*, 5:43–84, 2000.

[16] B. A. Nadel. Representation selection for constraint satisfaction: a case study using n-queens. *IEEE Expert: Intelligent Systems and Their Applications*, 5(3):16–23, 1990.

[17] J.-F. Puget. Symmetry breaking revisited. *Constraints*, 10(1):23–46, 2005.

[18] B. M. Smith. Symmetry and search in a network design problem. In *Proceedings of the 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2005.

[19] B. M. Smith. Modelling for constraint programming. In *ACP Summer School on Modelling with Constraints: Theory and Practice*, St Andrews, Scotland, UK, 2008.

[20] B. M. Smith, K. Stergiou, and T. Walsh. Modelling the Golomb ruler problem. Technical Report 1999.12, University of Leeds, UK, 1999.

[21] P. van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, 1999.

```
int: Weeks; int: GroupSize; enum Players = ...;
int: Groups=card(Players) div GroupSize;
assert Groups*GroupSize==card(Players): "invalid number of players";
predicate maxOverlap(list of var set of $E:sets,int:n)=
    forall(i,j in 1..length(sets) where i<J)
        (card(sets[i] intersect sets[j])=<n);
array[1..Weeks,1..Groups] of var set of Players: group;
constraint
    forall(i in 1..Weeks)
        (maxOverlap([group[i,j] | j in Groups],0));
constraint
    forall(i in 1..Weeks,j in 1..Groups)
        (card(group[i,j])==GroupSize);
constraint
    maxOverlap([group[i,j] | i in 1..Weeks,j in 1..Groups],1);
```

Figure 13: Zinc specification for the social golfer problem

```
cst weeks, groups, groupsize:N
dom $Players$=1..$groups$*$groupsize$, $Weeks$=1..weeks, $Groups$=1..groups;
var $Sched$:$(Players \times Weeks) \to^{groupsize \times weeks} Groups$
solve
    $\forall(h{:}Groups, w{:}Weeks)\, count(groupsize)$
        $(p{:}Players \mid Sched(p,w)=h) \wedge$
    $\forall(p_1 < p_2{:}Players)\, count(0..1)$
        $(w{:}Weeks \mid Sched(p_1,w)=Sched(p_2,w))$
```

Figure 14: ESRA specification for the social golfer problem

A Social golfer problem in other languages

Here we reproduce specifications for the social golfer problem in five other languages. These are taken from [11] and are in Zinc (Figure 13), ESRA (Figure 14), OPL (Figure 15), NP-SPEC (Figure 16) and ESSENCE (Figure 17: the unusual brackets are part of the language). These can be compared to the KOLMOGOROV specification in Figure 8.

```
int weeks=...; int groups=...; int groupsize=...;
range Weeks 1..weeks; range Groups 1..groups;
range Players 1..groups*groupsize;
var Groups Schedule[Players,Weeks];
subject to {
   forall(w in Weeks & g in Groups) (sum(p in Players)
      (Schedule[p,w]=g)=groupsize);
   forall(ordered p1,p2 in Players) (sum(w in Weeks)
      (Schedule[p1,w]=Schedule[p2,w])<2);
}
```

Figure 15: OPL specification for the social golfer problem

```
DATABASE
   weeks=6; groups=8; groupsize=4;
SPECIFICATION
   IntFunc({1..groups*groupsize}><{1..weeks},Schedule,1..groups).
   fail <- COUNT(Schedule(*,W,Gr),X), X!=groupsize.
   fail <- Schedule(P1,W1,Gr1), Schedule(P2,W1,Gr1), P1!=P2,
      Schedule(P1,W2,Gr2), Schedule(P2,W2,Gr2), W1!=W2.
```

Figure 16: NP-SPEC specification for the social golfer problem

```
language  ESSENCE 1.2.0
given     w, g, s : int(1..)
letting   golfers be new type of size g*s
find      sched:set(size w) of partition(numParts g, partSize s) from golfers
such that ∀ week1,weeks2∈sched . week1≠week2→
             ∀ group1∈parts(week1), group2∈parts(week2) . |group1∩group2| < 2
```

Figure 17: ESSENCE specification for the social golfer problem