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*
[2] *Department of Management, Cankaya University, Ankara, Turkey*
[3] *University of Edinburgh Business School, Edinburgh, UK*

Abstract. A theme of Algorithmic Information Theory is that, for any scientific or mathematical theory, "understanding is compression". That is, the more compactly we can express a theory, the more comprehensible it becomes. We apply this philosophy to the problem of constraint problem specification. Instead of defining a new specification language, we use Constraint Logic Programming as a meta-language to describe itself compactly via compression techniques. We show that this approach can produce short, clear descriptions of standard constraint problems. In particular, it allows a simple and natural description of compound variables and channeling constraints. Moreover, for problems whose specification requires the solution of an auxiliary problem, a single specification can unify the two problems. We call our specification language Kolmogorov.

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. Perhaps the best-known CP specification languages are OPL [13], Essence [11] and Zinc [15], while AMPL [10] is used to specify mathematical programs.

We propose a new approach to constraint problem specification, inspired by an idea from the field of Algorithmic Information Theory (AIT). G. Chaitin, one of its founders, has argued 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] and the view we discuss here is that *specification is compression*. Like scientific theories, constraint problem specifications should be concise and easily understood. In our approach specifications are *compressed constraint models* that can be derived by applying simple data compression-like techniques to constraint models. In principle any pattern in a model can be exploited to produce a more compact description, as in algorithmic data compression.

We choose Constraint Logic Programming (CLP) as both the problem modeling language and the specification language, with a few special predicates provided for its specification role. That is, we use CLP as a meta-language to describe CLP itself at a higher level of abstraction. We call our specification language KOLMOGOROV because of its connection with AIT and compression. A KOLMOGOROV specification can be automatically transformed (uncompressed) into an executable CLP model.

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. A version of this paper will be published in [18].

2 Specifications as compressed CLP models

KOLMOGOROV uses CLP in two ways: as a (high-level) specification language and as a (low-level) constraint modeling language (*low-level* is a relative term here because CLP is already quite a high-level language). In its high-level role it represents low-level concepts (variables, constraints and constants) as structured Prolog ground terms, and a few useful predicates are provided to aid description. We shall illustrate this by example. The specific CLP language we use is Eclipse [1] and we assume familiarity with basic CLP concepts.

2.1 Illustrative example: N-queens

Consider 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 [17] presented 9 constraint satisfaction problem (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 $\{1, \ldots, N\}$. 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 contains two queens. We need constraints to ensure that no column contains 2 queens: $v_i \neq v_j$ 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 (`kvar`), and the subgoals describing variable declarations and constraints (`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 but this could be done. The `[]` argument is used for passing parameters such as the size of the chess board, which is not used in this first specification.

```
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.
```

Fig. 1. A CLP model for 3-queens

To obtain the CLP variable declarations and constraints of the specification we enumerate all solutions of the goal

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

by backtracking:

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

These are collected into a list, then variable terms such as `v(2)` are replaced by unique CLP variable names `V1`, `V2` and `V3`. These names are generated automatically by (i) enumerating the variables via the goal

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

giving solutions

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

(ii) using a hash table to associate a unique integer (the hash table entry) with each ground term (the hash table key), and (iii) replacing each variable ground term by the letter `V` followed by its hash table entry. 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: for example we may 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\=J`. We can exploit this simple pattern to produce a more compact specification as shown in Figure 3. The `kvar` and `kgoal` solutions of this specification are the same as those of the previous one, and again exactly the same CLP model will be generated.

The predicate `csp` posts the constraints in its argument then nondeterministically solves the corresponding CSP by 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 KOLMOGOROV specifications.

```
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,[]).
```

Fig. 2. Naive KOLMOGOROV specification for 3-queens

```
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.
```

Fig. 3. Compressed KOLMOGOROV specification for 3-queens

We shall show that specifying variables and constraints as CSP solutions makes KOLMOGOROV very expressive.

The compact 3-queens specification can be generalised to the N-queens problem as shown in Figure 4, in which the board size `N` is specified in the second argument of `kvar` and `kgoal`. This specification is close to a mathematical description of the problem.

```
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.
```

Fig. 4. Compressed KOLMOGOROV specification for N-queens

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.

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.

2.2 Specification as compression

Exploiting patterns to obtain a more compact representation, as we did with the N-queens CLP model, 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 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. The compactness of the specification is limited only by the Kolmogorov complexity of the specification (hence the name KOLMOGOROV).

However, this principle should not be taken too far. We could in principle compress a constraint model to the shortest possible string $\mathcal{S}$, then write a KOLMOGOROV specification

```
kgoal(G,[]) :- uncompress(S,G).
```

This is not useful for our purposes because a random-looking string such as $\mathcal{S}$ is not comprehensible by humans, and in practice only compressions that improve clarity should be used.

3 Examples

We now present KOLMOGOROV specifications for several problems from the CP literature, and point 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 simple to extend KOLMOGOROV to optimisation problems.

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 [22], 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 [20] 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(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).
```

Fig. 5. KOLMOGOROV specification of the knapsack problem.

These specifications are of comparable size to ESSENCE and other specifications. For example specifications for the social golfer problem are shown in five other languages: ESSENCE, Zinc, ESRA, OPL and NP-SPEC. There is no generally-agreed way of comparing the relative sizes of specifications in such different languages, but to our eyes the KOLMOGOROV specification is no larger than any of the others, and smaller than some.

```
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,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).
```

Fig. 6. KOLMOGOROV specification of the Golomb ruler problem.

```
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((#(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.
kgoal((x(I,K)#=((I in r(K)) /\ (K in n(I)))),[N,M,S,R]) :-
   kvar(x(I,K),[N,M,S,R]).
```

Fig. 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.
```

Fig. 8. KOLMOGOROV specification of the social golfer problem.

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 [19]. We shall use this model to illustrate two powerful features of KOLMOGOROV: *symbolic constants* to simplify the writing of constraints, and incorporating an *auxiliary problem* (used to generate data for the main problem) into a specification.

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 represents all possible groups. (Puget notes that for the well-known case of 8 groups of 4 players over 10 weeks there are 35960 such groups, which is large but tractable.[1]) 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 for brevity, though our ordering of the `PL` variables breaks intra-group symmetry in the same way as Puget's use of set variables.

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 distinct term maps consistently 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, 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 separate phase, then added to the specification (or constraint model). Their

[1] Actually, our model is impracticably large in practice because we enumerate subsets. However, it is still a useful modelling exercise.

```
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,[_,_|_]).
```

Fig. 9. KOLMOGOROV specification for the improved social golfer

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 advantage of representing the elements of a CSP (variables, constants and constraints) as CSP solutions, we use a well-known industrial problem: the cutting stock problem. This example is taken from H. Kjellerstrand's MiniZinc page.[2]

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 allowed 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]).
```

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

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. Their generation is an *auxiliary problem* that must be solved before the cutting stock problem instance can be fully stated.

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

kgoal(c(P)::0..B,[B,PL,DL]) :-
   kvar(c(P),[B,PL,DL]).
kgoal(sum(L)#>=D,[B,PL,DL]) :-
   csp((Q::1..3,element(Q,DL,D))),
   findall(X*c(I),(element(I,PL,P),element(Q,P,X)),L).
```

Fig. 10. KOLMOGOROV specification for cutting stock

In real-world instances the set of allowed cutting patterns might 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 allowed cutting patterns. 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 as in Figure 11.

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

kgoal(c(P)::0..B,[B,DL]) :-
   kvar(c(P),[B,DL]).
kgoal(sum(L)#>=D,[B,DL]) :-
   csp((Q::1..3,element(Q,DL,D))),
   findall([U,V,W],csp(pattern(U,V,W)),PL),
   findall(X*c(I),(element(I,PL,P),element(Q,P,X)),L).

pattern(U,V,W) :-
   U::0..5, V::0..3, W::0..1, Used#=U*3+V*5+W*9,
   Waste#=17-Used, Used#=<17, Waste#<3.
```

Fig. 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* [21] and *channeling constraints* [5]. As an example we use a published CP model for covering arrays [14]. 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 [14] 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 for brevity). 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 [14]. However, an extensional method has the advantage of stronger filtering: 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 [14]). 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 constraint 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.

```
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).
```

Fig. 12. KOLMOGOROV specification for hybrid CA model

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. 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 [15], OPL [13], ESRA [9], $\mathcal{F}$ and Localizer [16], 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 when `kvar` is called from `kgoal` in several of our examples.

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. A particularly powerful feature is that the variables, constants and constraints of a CSP can all be specified as solutions to auxiliary CSPs. An auxiliary CSP might itself require specification, so a useful future direction is to make Kolmogorov recursive.

It might be argued 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 researchers. However, our approach requires the modeler to know only one language, whereas some 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 concision, 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.

Acknowledgment

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, M. Wallace. Constraint Logic Programming Using Eclipse. Cambridge University Press, 2007.
2. M. Balduccini. Representing Constraint Satisfaction Problems in Answer Set Programming. *ICLP'09 Workshop on Answer Set Programming and Other Computing Paradigms*, 2009.
3. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, 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, J. C. K. Wu. Increasing Constraint Propagation by Redundant Modeling: an Experience Report. *Constraints* **4**(2):167–192, 1999.
6. C. J. Colbourn, J. Zhou. Improving Two Recursive Constructions for Covering Arrays. *Journal of Statistical Theory and Practice* **6**:30–47, 2012.
7. F. Fages, J. Martin. From Rules to Constraint Programs with the Rules2CP Modelling Language. *Recent Advances in Constraints: 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, Lecture Notes in Computer Science* Vol. 5655, 2008, pp 66–83.
8. T. Fahle, S. Shamberger, M. Sellmann. Symmetry Breaking. *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2239, 2001, pp. 93–107.
9. P. Flener, J. Pearson, M. Ågren. Introducing ESRA, a Relational Language for Modelling Combinatorial Problems. *Proceedings of the 13th International Symposium on Logic Based Program Synthesis and Transformation*, 2003, pp. 214–232.
10. R. Fourer, D. Gay, 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, 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. *CP'01 Workshop on Symmetries*, 2001.
13. P. van Hentenryck. The OPL Optimization Programming Language. MIT Press, Cambridge, MA, 1999.
14. B. Hnich, S. D. Prestwich, E. Selensky, B. M. Smith. Constraint Models for the Covering Test Problem. *Constraints* **11**(3):199–219, 2006.
15. K. Marriott, N. Nethercote, R. Rafeh, P.J. Stuckey, M. Garcia de la Banda, M. Wallace. The Design of the Zinc Modelling Language. *Constraints* **13**(3):229–267, 2008.
16. L. Michel, P. van Hentenryck. Localizer. *Constraints* **5**:43–84, Kluwer Academic Publishers, 2000.

17. 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, IEEE Computer Society 1990.
18. S. D. Prestwich, S. A. Tarim, R. Rossi. Constraint Problem Specification as Compression. *2nd Global Conference on Artificial Intelligence*, 2016 (to appear).
19. J.-F. Puget. Symmetry Breaking Revisited *Constraints* **10**(1):23–46, 2005.
20. B. M. Smith. Symmetry and Search in a Network Design Problem. *Proceedings of the 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Lecture Notes in Computer Science* vol. 3524, 2005, pp 336–350.
21. B. M. Smith. Modelling for Constraint Programming. ACP Summer School on Modelling with Constraints: Theory and Practice. St Andrews, Scotland, UK, 2008.
22. B. M. Smith, K. Stergiou, T. Walsh. Modelling the Golomb Ruler Problem. Technical report 1999.12, University of Leeds, UK, 1999.