PKML with ClpZinc: A Compilation Chain from Search Strategies to Solvers, and its Use for Packing

Philippe Morignot, François Fages, and Thierry Martinez

Project-team LIFEWARE, Inria Paris-Rocquencourt, France

Abstract. Constraint programming is traditionally viewed as the combination of two components: a constraint model and a search procedure. On the other hand, packing seems to be a widely studied application of constraint programming. In this paper we present two languages: (i) ClpZinc, showing that tree search procedures can be fully internalized in the constraint model, and (ii) PKML a language for expressing symbolic packing relations among objects and their components (e.g., Allen relations between one-dimension shapes, symbolic relations between multi-dimension shapes, and symbolic relations among straight objects and their possible alternative shapes). The latter is implemented in the former, and reaches solvers through Zinc high-level models. PKML is evaluated on Korf's problem, improving the best performances [17] on several problem instances; it is also evaluated on an industrial problem with real data and complex spatial relations. Finally, the compilation chain from PKML to solvers is extended to real numbers.

1 Introduction

Constraint programming is traditionally presented as the combination of two components: a constraint model and a search procedure [18]. Front-end modeling languages are designed for solving problems using constraint programming solvers, thus either rely on a fixed strategy (e.g. Essence [6]), or contain special features for specifying the search strategy for the constraint solvers (e.g. Zinc [14]). The modeling language Zinc, and its implementation MiniZinc [1], succeeded in becoming a common input format across many solvers in the Constraint Programming community. In Zinc, the search procedure is specified through special annotations that are dedicated to the constraint solver [15] and ignored by the other solvers. The most recent language to express search strategies seems to be search combinators [16]. However ***

On the other hand, one of the many applications of constraint programming seems to be packing: how to pack boxes while respecting a given property (e.g., smallest englobing volume). A variant in 2 dimensions is to enclose non-overlapping squares of size 1x1 up to nxn in the smallest rectangle (Korf's problem [9,10]). Applications of packing include electronic design automation,

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

packing blocks in a circuit layout [12] and scheduling [13]. Due to the wideness of these applications, packing in itself has led to a large body of work, e.g., [8,2].

In this paper, we present two languages: (i) ClpZinc for expressing search strategies as constraints, which are then passed on to solvers as Zinc models (hence building the compilation chain "ClpZinc -> MiniZinc -> FlatZinc -> solvers"); And (ii) PKML (Packing Knowledge Modeling Language [5]) for expressing in ClpZinc symbolic relations among objects for packing (hence, augmenting the compilation chain to "PKML -> ClpZinc -> MiniZinc -> FlatZinc -> solvers").

The idea of ClpZinc is to use reified constraints to represent disjunctions ";" in the search strategy (expressed as CLP clauses) as boolean variables. These auxiliary variables are the basis for further constraints representing the search strategy. In other words, ClpZinc considers search not as a procedure but as constraint satisfaction [11].

PKML's performances are evaluated on Korf's problem and on an industrial one, both using the solver CHOCO [4] at the end of the toolchain. These performances successfully reach the best performances on the former problem [17]. Finally, an extension of the compilation chain to real numbers, and not only integers, is proposed, opening the way to interleaving finite-domain variables with real-interval variables. This extension is tested with the real-number interval-based solver IBEX [3] at the end of the chain.

The paper is organized as follows: section 2 presents an overview of the search strategy language ClpZinc (see [11] for a more detailed description). Then PKML is presented in section 3 and is evaluated on two problems; Section 4 describes the introduction of real numbers into the compilation chain. The last section sums up our contribution and opens issues for future work.

2 ClpZinc

The language ClpZinc is a Horn-based extension of Zinc where the item `solve satisfy;` in models is replaced by a CLP goal of the form "`:- goal.`", and where user-defined predicates are defined by CLP clauses of the form "`p(t1, ..., tn) :- goal.`".

For example, the following ClpZinc model implements the search strategy that enumerates all possible values for a given variable in ascending order.

```
var 0..5: x;
constraint x * x = x + x;

labeling(X, Min, Max) :-
   Min <= Max, (X = Min ; labeling(X, Min + 1, Max)).

:- labeling(x, 0, 5).
output [show(x)];
```

This ClpZinc model for the given goal of labeling `x` between 0 and 5, can be expanded to the following MiniZinc model:

```
var 0..5: x;
constraint x * x = x + x;
var 0..5: X1;
constraint X1 = 0 -> x = 1;
constraint X1 = 1 -> x = 3;
constraint X1 = 2 -> x = 5;
constraint X1 = 3 -> x = 4;
constraint X1 = 4 -> x = 2;
constraint X1 = 5 -> x = 0;
solve :: seq_search([
    int_search([X1], input_order, indomain_min, complete)
  ]) satisfy;
output [show(x)];
```

To be consistent with the search strategy of [17] (to which PKML's performances are compared, see section 3.2), the `interval_splitting` search strategy can be modelled:

```
% Succeeds when X is greater than or equal to Min and lesser than or
% equal to Max, enumerating the intervals of width Step between Min and Max.
interval_splitting(X, Step, Min, Max) :-
   Min + Step <= Max,
   NextX = min(X) + Step,
   ( X < NextX ; X >= NextX, interval_splitting(X, Step, Min + Step, Max) ).
interval_splitting(X, Step, Min, Max) :-
   Min + Step > Max.
```

3 Packing

The general problem of "packing" refers to stacking boxes while respecting some property, e.g., having the smallest volume, balancing weights, respecting a maximum distance among boxes. An example in 2 dimensions is the Korf's problem [9,10]: how to enclose squares of size 1x1 up to *nxn* in the smallest rectangle ? In this section, we propose a Packing Knowledge Modelling Language (PKML, initially proposed for the rule-based constraint programming language Rules2CP [5]) for ClpZinc, and evaluate its performances on Korf's problem and on an industrial one.

3.1 PKML

This language is composed of (i) symbolic relations over one-dimension shapes, i.e., Allen's relations over intervals [1], (ii) symbolic relations over multi-dimension shapes (Region Connection Calculus (RCC)), i.e., a multi-dimensional extension of Allen's relations; and (iii) symbolic relations over boxes/shifted boxes/shapes/objects. As represented in Fig. 1, a *shifted box* is a *box* shifted from the origin by

some vector; A *shape* is a set of contiguous *shifted boxes*; and an *object* is a set of possible alternative *shapes* (an *object* is a *shape* OR is another *shape* OR ... OR is a last *shape*, all shapes belonging to the set of alternatives). One application of this is that an *object* can represent rotation of *shapes*, i.e., an object o can include a first possible shape s_1 and a second possible shape s_2, where s_2 is a rotation of s_1 by 90 degrees for example, i.e., $o = s_1 \vee o = s_2$ (as in Fig. 1, d).

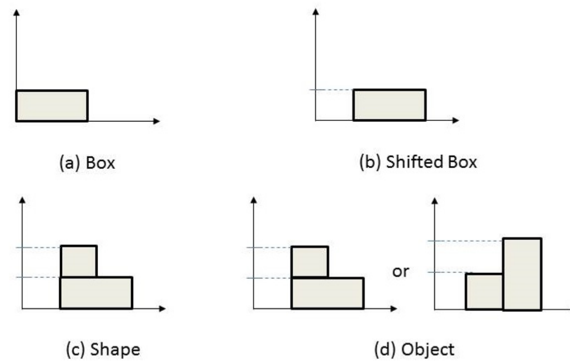

Fig. 1. PKML's boxes, shifted boxes, shapes and objects.

The following ClpZinc model represents Allen's symbolic relation of symetric non-overlapping and the multi-dimensional RCC version (i.e., two multi-dimensional tasks do not overlap *iff* they don't in at least one dimension):

```
% Allen
not_overlaps_sym(T1, T2, D) :-
   T2.start[D] >= T1.end[D] \/
   T1.start[D] >= T2.end[D].

% RCC
not_overlap_sym(T1, T2, N) :-
   exists (i in 1..N) ( not_overlaps_sym(T1, T2, i) ).
```

Here is the ClpZinc model of PKML expressing the internal representation of boxes, shifted boxes, shapes and objects:

```
% A box B is a list L of sizes in N dimensions
make_box(B, L) :-
    B = (sizes: L).

% A shifted box S is a box B (in N dimensions) and its offsets O in N dimensions
% with respect to the origin.
```

```
make_sbox(S, B, O) :-
   S = (box: B, offsets: O).

% A shape S is a list of shifted boxes SBs (in dimensions N).
make_shape(S, SBs) :-
   S = (sboxes: SBs).

% An object O is a list of alternative shapes Shapes, with its origin Origin (in
% dimension N) and weights Weight. Index is a list of indices of current alternative
% shapes in Shapes. e.g., [1] means that the first shape is the only choice.
make_object(O, Shapes, Index, Origin, Weight) :-
   O = (shapes: Shapes, shape_index: Index, origin: Origin, weight: Weight).
```

3.2 Example on Korf's problem

The implementation of PKML is first evaluated on Korf's problem, using (i) RCC (and the previous `not_overlap_sym` clause) and (ii) a more elaborate `non_overlapping` clause interfacing to the global constraint `geost` [2].

Here is the ClpZinc model which defines the n squares in the `geost` case — a square in PKML is an *object* composed of a unique *shape* defined by an un-shifted *box* of size NxN.

```
make_square(N, O) :-
   make_box(B, [N, N]),
   make_sbox(S, B, [0, 0]),
   make_shape(SH, [S]),
   make_object(O, [SH], [1], [0, 0], [100]).

make_squares(0, []).
make_squares(N, [O | Os]) :-
   N > 0,
   make_square(N, O),
   make_squares(N - 1, Os).
```

Fig. 2 plots all non overlapping versions' performances: an RCC version (noted "PKML RCC 1..200", 1 and 200 being the lower/upper bounds of the domain of the variables representing the squares' coordinates), a `geost` version (noted "PKML geost 1..200"), a re-implementation in ClpZinc of the version of [17] (noted "Simonis & O'Sullivan 1..200"), and the "naive" non-overlapping version (noted "Naive 1..200") using a 4-term disjunction for expressing non-overlapping of two squares and not using any search strategy. The `geost` and RCC versions use the same search strategy as [17] (including the `interval_splitting` one of section 2) except for the non-overlapping constraint, which is the global constraint `geost` in the former case and the ClpZinc clause `not_overlap_sym` in the latter.

The hardware on which this benchmark is performed on 2 Quad CPU at 2,83 GHz using 7,7 Gb RAM. This benchmark suggests that (i) the naive version

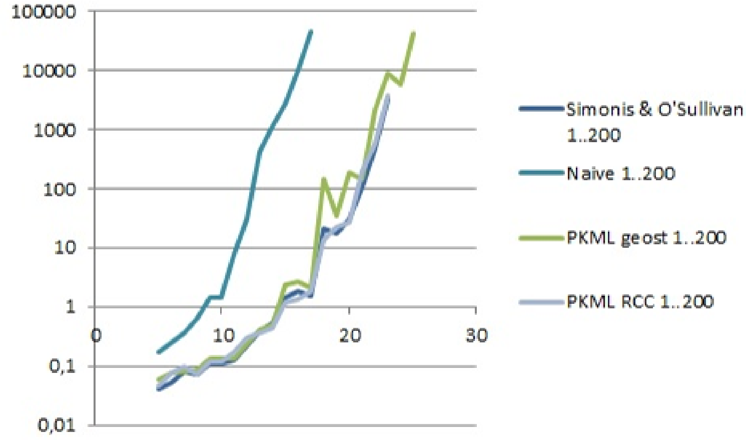

Fig. 2. Performances of PKML on Korf's problem: Computation time (in second) as a function of the size n of the largest square, on a logarithmic scale.

always exhibits the worst performances; (ii) the version of [17] exhibits the best performances (this version is the most efficient implementation of the Korf's problem so far), except on 5 problem instances; (iii) the `geost` version exhibits an overhead of 2 times on average when compared to [17]; (iv) the RCC version always exhibits better performances than `geost`, which is due to the overhead of the global constraint `geost` [2] and (v) the RCC version is comparable to [17], and reaches better performances on 5 problem instances (for n = 13, 14, 15, 16 and 18).

3.3 Example with real data

We now present a second example of PKML use, coming from the industry PSA. This example involves packing 10 *objects* with a unique *shape* and a weight, all using real data from this company.

Several spatial constraints are added to the model, due to its real aspect:

Gravity Any object is on the ground ($z = 0$), or there exists another object exactly below it;

```
% Post gravity constraint for objects Os and their corresponding tasks Ts.
gravity(Os, Ts) :- gravity(Os, Ts, []).
gravity([], _, _).
gravity([O|Os], [T|T1s], T2s) :-
   object_z(O, Z),
   Z = 0 \/ exists (t1 in T1s) (on_top(T, t1))
```

```
        \/ exists (t2 in T2s) (on_top(T, t2)),
    gravity(Os, T1s, [T|T2s]).
```

Weight stacking Objects are stacked by decreasing weight (the upper the lighter);

```
% Post the weight stacking constraint for objects Os and their
% corresponding tasks Ts
weight_stacking([], _).
weight_stacking([O|Os], [T|Ts]) :-
   weight_stacking_aux(O, Os, T, Ts), weight_stacking(Os, Ts).
   weight_stacking_aux(_, [], _, _).
weight_stacking_aux(O1, [O2|Os], T1, [T2|Ts]) :-
   not_above(T1, T2) \/ lighter(O1, O2),
   not_above(T2, T1) \/ lighter(O2, O1),
   weight_stacking_aux(O1, Os, T1, Ts).
```

Weight balancing Objects on the right of the scene and objects of the left of the scene must have their weight balanced, within some given percentage, e.g., 20%;

```
sum(Os, B, D, L, R) :- sum(Os, B, D, 0, L, 0, R).
sum([], _, _, L, L, R, R).
sum([O|Os], B, D, L1, L2, R1, R2) :-
   object_origin(O, D, S),
   object_end(O, D, E),
   object_end(B, D, BE),
   Left  = O.weight * bool2int(E <= floor(BE / 2)),
   Right = O.weight * bool2int(S >= floor(BE / 2)),
   sum(Os, B, D, Left + L1, L2, Right + R1, R2).

% Post the weight balancing constraint for objects Os, in scene-object B
% in dimension D and ratio Ratio
weight_balancing(Os, B, D, Ratio) :-
   sum(Os, B, D, L, R),
   100 * max(L, R) <= (100 + Ratio) * min(L, R).
```

Stack oversize The distance between the basis/tops of any pair of objects must be less than a given value, e.g., 20;

```
% Post the stack oversize constraint on tasks Ts, with a limit value of L
stack_oversize([], _).
stack_oversize([T|Ts], L) :- stack_oversize_aux(T, Ts, L), stack_oversize(Ts, L).

stack_oversize_aux(_, [], _).
stack_oversize_aux(T1, [T2|Ts], L) :-
    not_overlaps(T1, T2, 1) \/
    not_overlaps(T1, T2, 2) \/
```

```
    ( abs(T1.start[1] - T2.start[1]) <= L /\ abs(T1.end[1] - T2.end[1]) <= L /\
      abs(T1.start[2] - T2.start[2]) <= L /\ abs(T1.end[2] - T2.end[2]) <= L ),
    stack_oversize_aux(T1, Ts, L).
```

This problem is solved in PKML by the CHOCO solver in 0,451s.

4 Real Numbers

A main assumption of constraint programming is that variables own a finite (therefore enumerable) domain. In this section, we relax this assumption and extend variables' domain, and therefore the whole compilation chain, to real numbers.

4.1 In the compilation chain

For this, the interval-based real numbers solver IBEX [3], using contactors (playing a similar role for real variables as *domain reduction* for finite-domain variables), is integrated into CHOCO as a constraint. Since MiniZinc and FlatZinc already include real numbers (for connection with other solvers for linear programming and MILP), minimal patches (i.e., in the parsers and in the internal representation of variable) have been performed to ClpZinc and CHOCO to integrate real numbers.

Here is an example of ClpZinc model with real variables (note the dot "." denoting real numbers):

```
var 0.0..100.0: x;
var 0.0..100.0: y;
:- x + 10.0 = y.
output [ show(x) ++ show(y) ];
```

After compilation through the chain, here is the resolution using CHOCO+IBEX (note that IBEX contracts the real variables intervals due to the constraint above):

```
% parse instance...
length(int  vars) = 0
length(real vars) = 2
Activity-based
% solve instance...
x = (0.0; 90.0);
y = (10.0; 100.0);
----------
==========
% 1 Solutions, Resolution 0,001s (0,741586ms), 1 Nodes, 0 Backtracks,
0 Fails, 0 Restarts
```

4.2 Interleaving integers and reals

Interleaving integers and reals is always delicate — this is the problem in MILP for example. At the solver level (CHOCO+IBEX), the proposed global strategy is to apply search strategies on integers first, and then search strategies on real numbers — this is due to the greater domain reduction power of finite domain reasoning.

Integer variables and real variables can now be expressed in ClpZinc: The following ClpZinc model posts two constraints to set the input real variable `Y2` as the value of the output real variable `Y1` rounded to a (integer) grid composed of `N` ticks per unit interval. When `N` equals 1, this `grid` clause rounds the real variable `Y1` to the closest integer (as is performed in MILP), i.e., to variable `X` which is the integer counterpart of real output variable `Y2`.

```
% X is a temporary int variable. domain(X) = [LB(Y1) * N , UB(Y1) * N]
grid(Y1, Y2, X, N) :-
   abs(Y2 - Y1) < 1.0 / (2.0 * int2float(N)),
   Y2 = int2float(X) / int2float(N).
```

5 Conclusion

In this paper, we have presented two languages, ClpZinc for expressing search strategies, and PKML for expressing packing relations. Hence we built a compilation chain "PKML -> ClpZinc -> MiniZinc -> FlatZinc -> solvers". ClpZinc, a Horn-based modeling language, uses reified constraints to express disjunctions as boolean variables, leading to consider search as regular constraints. The advantages are easy expression of search strategies and more efficient resolution by solvers due to early propagation because of these additional constraints.

We also presented PKML, a language in ClpZinc for expressing symbolic relations among intervals (Allen's relations), multi dimensional objects (Region Connection Calculus) and objects with alternative possible shapes. Benchmarks of PKML on Korf's problem exhibit performances comparable to state-of-the-art ones, and even better on several problem instances. The PKML language also represents and solves a packing problem with real data and constraints. Finally, we presented an extension of this compilation chain to real numbers, i.e., real-interval variables and not only finite-domain variables, and a way to interleave both kinds of variables.

Future work includes using evolutionnary algorithms (e.g., CMA-ES [7]) to represent curved objects and apply them to packing with mixed curved/straight objects.

Acknowledgments

We are grateful to the French ANR for its support of the Net-WMS-2 project, and to the partners of this project for stimulating discussions on these topics.

References

1. J. Allen. Time and time again: The many ways to represent time. *International Journal of Intelligent System*, 6(4), 1991.
2. N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In *Proceedings of the 13th Conference on Pinrciples and Practice of Constraint Programming CP'07*, volume 4741 of *Lecture Notes in Computer Science*, pages 180–194, Providence, MA, USA, September 2007. Springer-Verlag.
3. Gilles Chabert and L. Jaulin. Contractor programming. *Artificial Intelligence*, 173:1079–1100, 2009.
4. École des Mines de Nantes. Choco 3, 2014.
5. François Fages and Julien Martin. From rules to constraint programs with the rules2cp modelling language. In *Proc. of Recent Advance in Constraints*, pages 66–83. Springer.
6. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.
7. Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.
8. Pascal Van Hentenryck. Scheduling and packing in the constraint language cc(fd). In M. Zweben and M. Fox, editors, *Intelligent Scheduling*, pages 1103 – 1108. Morgan Kaufmann Publishers.
9. Richard E. Korf. Optimal rectangle packing: Initial results. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *ICAPS*, pages 287–295. AAAI, 2003.
10. Richard E. Korf. Optimal rectangle packing: New results. In *ICAPS*, pages 142–149, 2004.
11. Thierry Martinez, François Fages, Philippe Morignot, and Sylvain Soliman. Search as constraint satisfaction. Technical Report TR-***, INRIA Rocquencourt, Le Chesnay, France, 2014.
12. Michael D. Moffitt, Aaron N. Ng, Igor L. Markov, and Martha Pollack. Constraint-driven floorplan repair. In Ellen Sentovich, editor, *DAC*, pages 1103 – 1108. ACM.
13. Michael D. Moffitt and Martha Pollack. Optimal rectangle packing: A meta-csp approach. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *ICAPS*, pages 93 – 102. AAAI.
14. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.
15. Reza Rafeh, Kim Marriott, Maria Garcia de la Banda, Nicholas Nethercote, and Mark Wallace. Adding search to zinc. In *CP*, pages 624–629, 2008.
16. Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J. Stuckey. Search combinators. *Constraints*, 18(2):269–305, 2013.
17. Helmut Simonis and Barry O'Sullivan. Search strategies for rectangle packing. In Peter J. Stuckey, editor, *Proceedings of CP'08*, volume 5202 of *LNCS*, pages 52–66. Springer-Verlag, 2008.
18. Pascal Van Hentenryck. *The OPL Optimization programming Language*. MIT Press, 1999.