Flexible, Rule-Based Constraint Model Linearisation

Sebastian Brand, Gregory Duck, Jakob Puchinger, Peter Stuckey

Flexible, Rule-based Constraint Model Linearisation

Sebastian Brand, Gregory J. Duck, Jakob Puchinger, and Peter J. Stuckey

NICTA, Victoria Research Lab, University of Melbourne, Australia

Abstract. Nonlinear constraint satisfaction or optimisation models need to be reduced to equivalent linear forms before they can be solved by (Integer) Linear Programming solvers. A choice of linearisation methods exist. There are generic linearisations and constraint-specific, user-defined linearisations. Hence a model reformulation system needs to be flexible and open to allow complex and novel linearisations to be specified. In this paper we show how the declarative model reformulation system Cadmium can be used to effectively transform constraint problems to different linearisations, allowing easy exploration of linearisation possibilities.

1 Introduction

The last decade has seen a trend towards high-level modelling languages in constraint programming. Languages such as ESRA [1], Essence [2], and Zinc [3] allow the modeller to state problems in a declarative, human-comprehensible way, without having to make subordinate modelling decisions or even to commit to a particular solving approach. Examples of decisions that may depend on the target solver are: the representation of variables of a complex type such as sets or graphs, and the translation of constraints into those provided by the solver to be used. Such decisions need to be taken if a concrete solver such as Gecode, ILOG Solver, CPLEX or Eclipse is to be used directly.

The problem solving process is thus broken into two parts. First, a high-level, solver-independent, conceptual model is developed. Second, the conceptual model is mapped to an executable version, the design model. Typically, an iterative process of solver selection, model formulation or augmentation, and model transformation, followed by experimental evaluation, is employed.

An imbalance exists in how the steps of this process are supported in practice. For the task of model formulation, there are well-designed, open, high-level modelling languages. In contrast, the task of model transformation is typically done by fixed procedures inaccessible to the modeller. It is hard to see that there is a single best set of transformations that can be wrapped and packed away. We therefore conclude that a strong requirement on a model transformation process and platform is *flexibility*.

In this paper we describe how we transform high-level models written in the modelling language MiniZinc [4] (a subset of Zinc) into Integer Linear Programming (ILP) models. The transformations are written in our term-rewriting

based model transformation language CADMIUM [5]. The rules and transformations are directly accessible to the modeller and can be freely examined, modified, and replaced. A major strength of CADMIUM is its tight integration with the ZINC modelling language. The rules operate directly on ZINC expressions; as a result, transformations are often very compact and comprehensible. Another strength of the approach is easy reusability. For example, in the linearisation of MINIZINC models we reused transformations originally designed for transforming MINIZINC models to FLATZINC (a low-level CP solver input language).

Our computational experiments, where MINIZINC models are transformed into CPLEX LP format, demonstrate the advantages of our system. It allows the user to experiment with different ways of linearising logical constraints as well as high-level constraints such as `all_different` [6, 7].

2 Languages and systems

2.1 The ZINC family of modelling languages

ZINC [3] is a novel, declarative, typed constraint modelling language. It provides mathematical notation-like syntax (arithmetic and logical operators, iteration), high-level data structures (sets, arrays, tuples, Booleans), and extensibility by user-defined functions and predicates. Model and instance data can be separate. MINIZINC [4] is a subset of ZINC closer to existing CP languages that is still suitable as a medium-level constraint modelling language. FLATZINC, also described in [4], is a low-level subset of ZINC designed as a CP solver input language. It takes a role for CP systems comparable to that taken by the DIMACS and LP/MPS formats for propositional-satisfiability solvers and linear solvers, resp.

A ZINC model consists of an unordered set of *items* such as variable and parameter definitions, constraints, type definitions, and the solving objective. As an example, consider the following MINIZINC model of the Golomb Ruler problem. The problem consists in finding a set of small integers of given cardinality such that the distance between any pair of them differs from the distance between any other pair.

```
int: m = 4;
int: n = m*m;
array[1..m] of var 0..n: mark;
array[1..(m*(m-1)) div 2] of var 0..n: differences =
    [ mark[j] - mark[i] | i in 1..m, j in i+1..m ];

constraint mark[1] = 0;
constraint % The marks are ordered, and differences distinct
    forall ( i in 1..m-2 ) ( mark[i] < mark[i+1] )
    ∧ all_different(differences);
constraint mark[2] - mark[1] < mark[m] - mark[m-1];   % Symmetry

solve minimize mark[m];
```

Let us consider the items in textual order.

– The first and second lines declare the parameters `m` and `n`, both of type `int`.
– The following two lines declare the decision variables in arrays `mark` and `differences`. The variables of either array take integer values in the range 0..`n`. The index set of `mark` are the integers in the range 1..`m`. The array `differences` is defined by an array comprehension.
– Next is a *constraint* item fixing the first element of `mark` to be zero. The remaining constraints order the marks, make the differences distinct, and finally break a symmetry.
– The final item is a *solve* item, which states that the optimal solution with respect to minimising the final mark at position `m` should be found.

More detail about the Zinc language family is available in [3, 8, 4].

2.2 The Cadmium model transformation system

Cadmium [5] is a declarative, rule-based programming language based on associative, commutative, distributive term rewriting. Cadmium is primarily targetted at Zinc model transformation, where one Zinc model is transformed into another by a Cadmium program (mapping). A rule-based system for constraint model transformation is a natural choice as such transformations are often described as rules in the first place.

Cadmium is well-suited for Zinc model transformation because of the tight representational integration between the two languages. A Cadmium program is a sequence of rules of the form

$$\mathit{CCHead} \setminus \mathit{Head} \Leftrightarrow \mathit{Guard} \mid \mathit{Body}$$

where *Head* and *Body* are arbitrary terms that in particular can involve Zinc expressions. Any expression from the current model matching *Head* is rewritten to the expression *Body* if the rule application requirements given by *CCHead* and *Guard* are satisfied (either of which can be absent). The rules in the program are repeatedly applied until no more applications are possible. The obtained model is the result of the transformation.

Cadmium has its roots in `CHR` [9] but substantially extends it by several features, briefly described in the following. See [5] for a thorough exposition.

Associative commutative matching. An operator $\circ$ is *Associative Commutative* (AC) if it satisfies $x \circ (y \circ z) = (x \circ y) \circ z$ and $x \circ y = y \circ x$. AC operators are common, e.g. $+$, $*$, $\wedge$, $\vee$, $\cup$, $\cap$. Cadmium supports *AC matching*, which means the order and nested structure of expressions constructed form AC operators does not matter; e.g. `0 + a` can match `X + 0` with `X = a`. This reduces the number of rules required to express a transformation. AC matching is a standard feature of other term rewriting languages, e.g. Maude [10].

Conjunctive context matching. CADMIUM supports *matching* based on the pseudo-distributive property $X \wedge f(Y_1, ..., Y_n) = X \wedge f(Y_1, ..., X \wedge Y_i, ..., Y_n)$ of conjunction for all functions f. This is in contrast to *performing* classical distribution where the X disappears from the top-level and is distributed to all arguments at once. Using this approach, conjunct X is *visible* in any sub-expression S of f: we say that X is in the conjunctive context (CC) of S.

A CADMIUM rule in which a *CCHead* prefix is present uses CC matching. In order for the rule to fire, *CCHead* must match (part of) the conjunctive context of the expression that matches *Head*. CC matching can for example be used to implement parameter substitution in constraint models by the rule

`X = C \ X` $\Leftrightarrow$ `C`.

If an equation `X = C` appears in the conjunctive context of an `X`, then this rule rewrites `X` to `C`. Consider the expression `f(a,a+b,g(a))` $\wedge$ `a = 3`. Equation `a = 3` is in the CC of all occurrences of `a` in the rest of the expression. After exhaustively applying the rule, the result is `f(3,3+b,g(3))` $\wedge$ `a = 3`.

CC matching is very powerful because it allows the user to match against non-local information. As far as we are aware, CC matching is unique to CADMIUM.

User-definable guards. CADMIUM supports rule with guards. A rule in which a guard is present can only be applied if the guard holds, that is, if the *Guard* expression can be rewritten to `true`. CADMIUM provides a number of simple guards, such as `is_int(X)` to test whether `X` is an integer constant. Importantly, guards can also be defined by the user via rules.

Staged transformations. Beyond atomic transformations that consist of a single rule set, CADMIUM also supports composite, staged transformations: sequences of atomic transformations. Each atomic transformation is applied to the model in sequence, with a commitment to the intermediate results.

2.3 Representation of ZINC models in CADMIUM

Conceptually, CADMIUM operates directly on ZINC expressions and items (we emphasise this by printing ZINC keywords in bold). The following details of the ZINC representation in CADMIUM term form are worth pointing out:

- All ZINC items in the model are joined by conjunction. Thus the ZINC model
  constraint `X = 3;`
  solve satisfy`;`
  is treated as
  constraint `X = 3` $\wedge$ **solve satisfy**`.`
  The advantage is that ZINC items are in each other's conjunctive context.
- The conjunction of ZINC items is wrapped by a top-level **model** functor. This representation allows top-down model transformation in the way non-term-rewriting-based approaches work, rewriting the entire model at once:
  model `Model` $\Leftrightarrow$ `...`
  However, in our experience, top-down model transformations are almost never needed.

3 Transforming nonlinear MiniZinc into linear Integer Programming format

There are several ways of linearising constraints. A generic method is the *Big-M* approach, used to convert a logical combinations of linear constraints into a conjunction of linear constraints. A finite domain constraint can always be written as a logical combination of linear constraints, by reverting to some logical definition of it.

For some high-level constraints, alternative definitions can be given that tightly reflect their structure onto auxiliary variables, for example, 0/1 integer variables encoding assignments of original variables.

3.1 The generic *Big-M* transformation

At the core of this linearisation approach is the fact that a disjunction $(x \leqslant 0) \vee b$, where b is a propositional variable, is equivalently written as the inequation $x \leqslant \mathit{ubound}(x) \cdot b$, where *ubound* is an upper bound on the value of the variable x. Our transformation first normalises a MiniZinc model and then transforms it into negation normal form. The next steps are based on the work by McKinnon and Williams [6] and Li et al. [11]. We simplified their transformation and made some steps, such as Boolean normalisation, more explicit.

Li et al. [11] define the modelling language $\mathcal{L}^+$, which consists of linear arithmetic constraints, Boolean operators, and some additional constraints such as `at_most` and `at_least`. Steps of the transformation described in [11] are:

- Transformation of $\mathcal{L}^+$ into negation normal form.
- Transformation of simplified $\mathcal{L}^+$-formulas into Γ-formulas. A Γ-formula is of the form $\Gamma_m\{P_1, \ldots, P_n\}$ and is true if at least m of $\{P_1, \ldots, P_n\}$ are true. Each P_i is a Γ-formula, a linear constraint, or a propositional literal.
- Flattening of nested Γ-formulas.
- Transformation of Γ-formulas into linear constraints.

Our transformation is based on this procedure. After several normalisation and decomposition steps, we generate Γ-formulas which are then further transformed into a linear form of MiniZinc. In the decomposition steps we provide several alternative transformations, and we allow the user to experiment with possible combinations of those alternatives. As a final step, we currently write out the obtained linear model in CPLEX LP format, for directly feeding it into most of the currently available ILP solvers.

We outline the major transformation steps in the following, giving actual Cadmium example rules for illustration.

Model normalisation. MiniZinc allows substantial freedom in the way models are written and so adapts to the preferred visual style of the model writer. The first step in our conversion is to rewrite simple, equivalent notations into a normal form. Examples are the joining of constraint items and the replacement of synonyms:

```
(constraint C) ∧ (constraint D) ⇔ constraint C ∧ D;
X == Y ⇔ X = Y;
```

Predicate inlining. We currently use a top-down transformation, traversing the entire model term, to replace a call to a predicate (or function) by the respective instantiated predicate body.

This is our only case of a model-wide top-down transformation. We are currently moving towards a modified ZINC term representation in which predicate applications are wrapped in a reserved functor. Matching can then take place against this functor, and the need for a top-down transformation will be removed.

Parameter substitution and comprehension unfolding. The next steps, while defined separately and listed in sequence, depend on and enable one another. In a non-term-rewriting approach, an explicit iteration loop would be needed to compute the mutual fixpoint. In CADMIUM, each individual atomic transformation corresponds to a set of rules, and the composite transformation is the union of these rule sets. Once the composite transformation has reached stabilisation, the mutual fixpoint of the separate rule sets is obtained.

1. Parameter substitution.
   We use the conjunctive context of a parameter to retrieve its value:
   ```
   X = C \ X ⇔ is_int(C) | C;
   ```
2. Evaluation.
   Parameter substitution may allow us to simplify the model. We here apply rules that do so by taking into account the semantics of ZINC constructs:
   ```
   X ∨ true ⇔ true;
   X + Y ⇔ is_int(X) ∧ is_int(Y) | X !+ Y;
   X ⩽ C ⇔ is_int(C) ∧ ubound(X) !⩽ C | true;
   ```
   The first rule simplifies a Boolean expression, the second evaluates addition of integer constants using the CADMIUM built-in `!+`, while the third removes constraints $X \leqslant C$ that are redundant w.r.t. to the declared domain of X.
3. Compound built-in unfolding.
   This step inlines predicates/functions such as **forall**, **sum** that are compound built-ins in MINIZINC:
   ```
   sum([]) ⇔ 0;
   sum([E ! Es]) ⇔ E + sum(Es);
   ```
   Note the CADMIUM syntax for array literal decomposition shown here.
4. Comprehension unfolding.
   An example for a simple case are these rules:
   ```
   [E | X in L..U] ⇔ L > U | [];
   [E | X in L..U] ⇔ [subst(X=L, E) ! [E | X in L+1..U]];
   ```
   The `subst` term denotes a substitution and is reduced accordingly.

These transformations are not specific to the MINIZINC linearisation task. Indeed, they are also used in the MINIZINC to FLATZINC transformation.

Decomposition of high-level constraints. In addition to the previously defined normalisations and decompositions, we decompose different generic constraints such as the `domain` constraint, here in ZINC notation:

```
X in A..B ⇔ is_int(A) ∧ is_int(B) | A ⩽ X ∧ X ⩽ B;
X in S    ⇔ is_set(S) | exists([ X = D | D in S ]);
```

We discern two cases of the respective set. If it is in range form, the constraint can be mapped onto two inequalities. Otherwise, it is mapped to a disjunction over the set values, which can be written using ZINC comprehension notation.

An array lookup with a variable index, corresponding to an `element` constraint, is transformed into a disjunction over all possible index values:

```
Y = A[X] ⇔ is_variable(X) |
    exists([ X = D ∧ A[D] = Y | D in dom(X) ]);
```

The expression **dom**`(X)` using the ZINC built-in **dom** is rewritten into the declared domain of the variable `X`, by rules we omit here. ZINC has a variety of such built-ins; **index_set** to retrieve an array index set is another useful one.

An `all_different` constraint is simply decomposed into a conjunction of inequations of the variable pairs:

```
all_different(X) ⇔
    forall([ X[I] ≠ X[J] | I,J in index_set(X) where I < J ]);
```

Minimum and maximum constraints are similarly decomposed. Furthermore, strict inequalities and disequalities are rewritten into expressions using only inequalities.

Since the decomposition of high-level constraints may introduce comprehensions and since further expression simplification can often be done, the rules for comprehension unfolding and expression evaluation are again imported into this stage.

Negation normal form. We transform formulas into negation normal form in the usual way. For example

```
(x - y < 5 ∧ y - x < 5) → (z ⩾ 1)
```

is rewritten into

```
((x - y ⩾ 5) ∨ (y - x ⩾ 5)) ∨ (z ⩾ 1).
```

***N*-ary conjunction and disjunction.** We conjoin these binary connectives into an n-ary form, (using functors `conj,disj`), which is then transformed into Γ-formula form:

```
disj(Cs) ⇔ gamma(Cs, 1);
conj(Cs) ⇔ gamma(Cs, length(Cs));
```

The second argument to `gamma` is the minimum number of subformulas that need to hold. The formula from the example above becomes:

```
gamma([gamma([x - y ⩾ 5, y - x ⩾ 5], 1), z ⩾ 1], 1).
```

***Big-M* linearisation.** This is the central step. It relies on the fact that all constraints were previously normalised. We proceed top-down, starting at the top-most `gamma` formula.

```
constraint gamma(Cs, M) ⇔
    constraint implied_gamma(true, Cs, M, []);

implied_gamma(B, [], M, Bs) ⇔ B → sum(Bs) ⩾ M;
implied_gamma(B, [C ! Cs], M, Bs) ⇔
    let { var bool: Bi } in
    ((Bi → C) ∧ implied_gamma(B, Cs, M, [bool2int(Bi) ! Bs]));

B → E ⩾ F ⇔ E-F ⩾ lbound(E-F) * (1-bool2int(B));
```

The second and third rule transform a formula $B \rightarrow \Gamma_m(C)$ into a conjunction of implications $B_i \rightarrow C_i$. The B_i are accumulated in a list, which is used for the constraint $B \rightarrow \sum_i B_i \geqslant m$. An implication whose consequence is a `gamma` formula is turned into `implied_gamma` form again (not shown here for brevity). The last rule finally rewrites a simple implied linear inequation into pure linear form. The `lbound` term is rewritten into a safe lower bound of its argument expression which may include decision variables.

We optimise the linearisation by distinguishing special cases such as in

```
implied_gamma(B, [Bi ! Cs], M, Bs) ⇔ is_variable(Bi) |
    implied_gamma(B, Cs, M, [bool2int(Bi) ! Bs]);
```

which leads to fewer auxiliary Boolean variables being created.

Let us revisit part of our example. Assume x and y are in 0..10.

```
gamma([x - y ⩾ 5, y - x ⩾ 5], 1)
```

is stepwise transformed as follows (where we omit **bool2int** for brevity):

```
B → gamma([x - y ⩾ 5, y - x ⩾ 5], 1)

implied_gamma(B, [x - y ⩾ 5, y - x ⩾ 5], 1)

(B1 → x - y ⩾ 5) ∧ (B2 → y - x ⩾ 5) ∧ (B → B1+B2 ⩾ 1)

(x - y - 5 ⩾ -15*(1 - B1)) ∧ (y - x - 5 ⩾ -15*(1 - B2)) ∧
       (B1 + B2 - 1 ⩾ -1*(1 - B))
```

Boolean variables to 0/1 variables. In this step, we recast Boolean variables as 0/1 integer variables, by simply substituting the type:

```
bool ⇔ 0..1;
bool2int(B) ⇔ B;
```

Output to LP format. The concluding stage prints out the linear model in CPLEX LP format using CADMIUM's I/O facilities. The result of applying the transformations to the Golomb Ruler problem of Section 2.1 is given in the Appendix.

3.2 Equality encoding for high-level constraints

For a constraint such as `all_different`, the *Big-M*-linearisation applied to its naive decomposition does not result in a so-called *sharp* formulation: one that represents the convex hull of the constraint. Sharp formulations for a number of common constraints are given in Refalo [7]. At the core of many sharp formulations is the explicit encoding of variable assignments. Given a variable x with domain $D(x)$, for each $a \in D(x)$ a propositional variable for the assignment $x = a$ is introduced. We write such a variable as $[\![x = a]\!]$.

In this way, a sharp linear formulation of the `domain` constraint $x \in S$ is

$$\sum_{a \in D} [\![x = a]\!] = 1 \quad \wedge \quad x = \sum_{a \in D} a[\![x = a]\!].$$

For the `all_different` constraint over variables x_i with respective domain $D(x_i)$, one can use the linear constraints

$$\sum_{i=1}^{n} [\![x_i = a]\!] \leqslant 1 \qquad \text{for each} \quad a \in \bigcup_{i=1}^{n} D(x_i).$$

They represent the fact that each value in any variable domain can be used by at most one variable. The CADMIUM rule setting up this encoding is as compact:

```
all_different_equality_encoding(Xs, Xi_eq_a) ⇔
    forall([ sum([ Xi_eq_a[I,A] | I in index_set(Xs) ]) ⩽ 1
        | A in array_union([ dom(Xs[I]) | I in index_set(Xs) ]) ]);
```

The array `Xi_eq_a` collects the ⟦ x = a ⟧ variables. In order to share these encoding variables between different high-level constraints, the link between an original variables x and its associated encoding variables is maintained by encoding tokens (terms). These tokens are installed at the model top-level during the encoding stage and are thus in the conjunctive context of any constraint whose translation needs them.

To contrast the available approaches for `all_different`, consider the MINIZINC fragment:

```
array[1..n] of var -n..n: x;
constraint all_different(x);
```

The *Big-M* translation of `all_different` gives:

```
array[1..n, 1..n] of 0..1: B1;
array[1..n, 1..n] of 0..1: B2;
constraint
    forall(i in 1..n, j in i+1..n) (
        x[i] - x[j] + 1 ⩽ (2 * n + 1) * (1 - B1[i, j]) ∧
        x[j] - x[i] + 1 ⩽ (2 * n + 1) * (1 - B2[i, j]) ∧
        B1[i, j] + B2[i, j] ⩾ 1 );
```

while the transformation using the equality encoding results in:

```
array[1..n, -n..n] of 0..1: xv;
constraint
    forall(i in 1..n) (
        sum([ a * xv[i, a] | a in -n..n ]) = x[i] ∧
        sum([     xv[i, a] | a in -n..n ]) = 1 );
constraint
    forall(a in -n..n) ( sum([ xv[i, a] | i in 1..n ]) ⩽ 1 );
```

The `element` constraint $z = a[x]$ for a variable x and an array a of integer constants can be represented as

$$z = \sum_{i \in D(x)} a[i] \cdot [\![x = i]\!],$$

which is embodied in the rule

```
element_equality_encoding(A, X, Y, X_eq_d) ⇔
    Y = sum([ A[D] * X_eq_d[D] | D in dom(X) ]);
```

This encoding is not applicable in the case when the array has variable elements. Our transformation verifies this and falls back to the naive *Big-M* decomposition approach if needed.

The basis for these linearisations of high-level constraints comes from the linear representation of disjunctive programs [12]. A further generalisation would be to directly apply this linearisation to the constraint in negation normal form.

3.3 Context-dependent constraint generation

If we take into account the context of a constraint we may be able to simplify its translation. The Tseitin transformation [13] for converting Boolean formulas into clausal form takes this into account, usually reducing the number of clauses by half. For (Integer) Linear Programming there are common modelling "tricks" that make use of context. For example, the `max(y,z)` expression in both of the following cases

```
constraint 8 ⩾ max(y,z);
solve minimize max(y,z);
```

can be replaced by a new `x` constrained by `x ⩾ y ∧ x ⩾ z`. In the first case, we only need to require the existence of any value between 8 and `y,z`, and in the second case, minimisation will force `x` to equal either `y` or `z`.

In general if a variable is only bounded from above in all constraints, we can translate an equation defining the variable as an inequality that bounds it from below. For example `x = max(y,z)` is replaced by `x ⩾ y ∧ x ⩾ z` as above if `x` is only bounded from above, and replaced by `x ⩽ t ∧ (t ⩽ y ∨ t ⩽ z)`, where `t` is a new variable, if `x` is only bounded from below.

This reasoning can be concisely implemented in rule form:

```
max(X, Y) ⇔ pol(ID, pos, max_context(X,Y, ID));

E + pol(ID, P, F) ⇔ pol(ID, P, E + F);
E - pol(ID, P, F) ⇔ pol(ID, invert(P), E - F);

pol(ID, P, E) ⩽ F ⇔ pol(ID, P, E ⩽ F);
pol(ID, P, E) ⩾ F ⇔ pol(ID, invert(P), E ⩾ F);
pol(ID, _, E) = F ⇔ pol(ID, all, E = F);

constraint pol(ID, P, E) ⇔ pol(ID, P) ∧ constraint E;
solve minimize pol(ID, P, E) ⇔ pol(ID, P) ∧ solve minimize E;

pol(ID,all) \ max_context(X,Y,ID) ⇔ max_complete(X,Y);
pol(ID,pos) \ max_context(X,Y,ID) ⇔ max_bounded_above(X,Y);
```

We add a polarity marker to each occurrence of a nonlinear expression in question. Polarity markers then travel upwards in the expression tree until the top-level, recording possible polarity changes. (The rules for `invert`, not shown here, map `pos` to `neg` and vice versa, and `all` to itself). Once at the top-level, the polarity of the expression occurrence is known, and it can be replaced accordingly.

3.4 Constraint relaxations

Given we have completely captured the meaning of a high-level constraint such as `element` or `all_different` by some linearisation, we are free to add other linear relaxations of the constraints to the model in order to improve the solving behaviour. Hooker [14] describes a number of simple and complex linear relaxations for various high-level constraints.

As an example, consider the `element` constraint $Y = A[X]$ where A is a fixed array. We can add bounds to Y as follows:

```
Y = A[X] ⇔ is_variable(X) ∧ fixed_array(A) |
    exists([X = D ∧ A[D] = Y | D in dom(X)]) ∧
    Y ⩾ min([A[D] | D in dom(X)]) ∧
    Y ⩽ max([A[D] | D in dom(X)]);
```

4 Case studies

In this section we report on evaluations of various choices in transforming MiniZinc into LP format. We show that the best choice is problem-dependent and, therefore, that an open transformation system facilitating experimentation is important. For reference, we also give results on transforming MiniZinc to the low-level CP solver input language FlatZinc.

The experiments were performed on a 3.4 Ghz Intel Pentium D with 4 Gb RAM computer running Linux. The FlatZinc models were solved by the G12 finite domain solver using its default (first-fail) search. The LP models were solved using CPLEX 10.0 with default parameter settings. The solvers were aborted if they did not return a result within a reasonable amount of time; this is indicated in the tables.

Table 1. Results of the described transformations on several different models

name	MiniZinc	FlatZinc			LP *Big-M* decomp.			LP equality enc.		
	lines	lines	transl.	solve	lines	transl.	solve	lines	transl.	solve
eq20	63	82	0.31s	0.18s	43	0.44s	0.00s	〃		
jobshop2x2	20	18	0.28s	0.10s	37	0.40s	0.00s	〃		
jobshop4x4	22	141	0.31s	0.18s	227	0.48s	0.02s	〃		
jobshop6x6	24	492	0.49s	8.65s	749	0.67s	1.72s	〃		
jobshop8x8	26	1191	0.73s	>300s	1771	1.11s	>300s	〃		
mdknapsk5_3	21	16	0.29s	0.07s	25	0.42s	0.00s	〃		
mdknapsk100_5	75	176	0.60s	>300s	217	1.36s	0.61s	〃		
packing	32	237	0.33s	0.16s	378	0.53s	0.00s	〃		
queens_8	9	86	0.31s	0.17s	613	0.56s	0.06s	〃		
queens_10	9	137	0.32s	0.15s	974	0.72s	0.36s	〃		
queens_20	9	572	0.49s	0.21s	4039	2.42s	>300s	〃		
alpha	52	53	0.29s	0.16s	2356	1.64s	0.13s	1511	1.32s	0.51s
golomb4	11	14	0.30s	0.07s	144	0.46s	0.00s	272	0.47s	0.01s
golomb6	11	25	0.31s	0.18s	807	0.69s	0.10s	1249	1.02s	0.53s
golomb8	11	40	0.32s	1.49s	2763	1.70s	19.36s	3878	3.28s	>300s
perfsq10	16	89	0.28s	0.17s	949	0.91s	0.12s	493	0.60s	0.10s
perfsq20	16	161	0.30s	1.55s	3069	3.36s	1.92s	1353	1.14s	0.42s
perfsq30	16	233	0.29s	111.29s	6389	9.10s	21.00s	2613	2.34s	0.66s
warehouses	45	476	0.45s	2.29s	1480	1.14s	1.34s	1322	0.96s	0.08s

4.1 *Big-M* decomposition and equality encoding

For this comparison we use the following examples:

- eq20: twenty linear constraints;
- jobshop: square job scheduling (2×2, 4×4, 6×6, 8×8);
- mdknapsk: multidimensional knapsack problem ($\langle n, m \rangle \in \{\langle 5, 3\rangle, \langle 100, 5\rangle\}$);
- packing: packing squares into a rectangle (size 4);
- queens: the N-queens problem (sizes 8, 10, 20);
- alpha: a crypt-arithmetic puzzle;
- golomb: the Golomb ruler problem ($m \in \{4, 6, 8\}$);
- perfsq: find a set of integers whose sum of squares is itself a square (maximum integer 10, 20, or 30);
- warehouses: a warehouse location problem.

The results are shown in Table 1. The problems are grouped according to the translation features they can make use of. The eq20 and mdknapsk problems are linear and used to gauge the performance of the parts of the transformation not concerned with linearisation as such. The job-shop, packing and queens problems are nonlinear models without the use of high-level constraints, so the equality encoding variant does not differ from the *Big-M* variant for them. The alpha and golomb problems use `all_different` constraints, whereas `element` constraints occur in perfsq and warehouses.

First, from these experiments we can see that while the FlatZinc translations are often smaller, and faster to achieve than the LP format, the speed of the ILP solver means that the LP translations are often better overall.

That the translation to LP is typically slower than to FlatZinc is not unexpected as linearisation creates more constraints. A second, central factor is that, while FlatZinc is output by a non-Cadmium Zinc pretty printer, the LP format generator uses a preliminary, primitive Cadmium I/O module to write the files. We plan to address this issue by passing the linear model to the ILP solver directly rather than via files; and we will also optimise Cadmium I/O.

Some of the slightly bigger examples (golomb8, jobshop8x8, mdknapsk100_5, perfsq30, and queens_20) show that translations times do scale, but the solve times can increase dramatically. For some examples (queens, golomb, jobshop) we can see a clear advantage of the FD solver, whereas for other examples (mdknapsk, perfsq) the ILP solver performs better.

For the linearisation choice, we find that for our example problems the sharp equality encodings works well for `element`, whereas surprisingly `all_different` does not benefit from it.

4.2 Context-dependent `max` constraints

For this set of benchmarks we use a model of a cancer radiation therapy problem [15]. The model is linear with the exception of multiple occurrences of `max` constraints. We compare the generic, complete linearisation and the context-dependent one (Section 3.3). Table 2 shows the results.

One observation is that the LP translation time grows quickly with the instance size. In good part this is due to Cadmium's current suboptimal I/O module: for example, approximately one third of the time for size 8 instances is spent in the final step of printing the LP format text file.

The major observation in these benchmarks results, however, is the very surprising fact that the complete linearisation is *better* in the majority of cases than the context-dependent translation, which is less than half the size. This appears to be a consequence of an ill-guided ILP search in CPLEX in the context-dependent case. While correct bounds on the solutions are often found quickly, the search struggles to find integer solutions. We have been able to drastically improve the behaviour for some instances by an informed modification of CPLEX parameters.[1]

A study of this unexpected observation is not the task of this paper. This puzzling result does, however, support our claim that the flexibility to experiment with different model translations is important.

5 Concluding remarks

Cadmium is one of only a few purpose-built systems targetting constraint model transformation, and among these, has particular strengths. Constraint Handling

[1] Interestingly an FD approach outperforms ILP when a specialised search approach is used [15], but this is not currently expressible in MiniZinc.

Table 2. Radiation problems: generic and context-dependent translations

Instance	MiniZinc	FlatZinc		LP complete			LP context-dependent		
	lines	lines	transl.	lines	transl.	solve	lines	transl.	solve
8_0	12	2387	3.20s	7458	16.30s	5.86s	2530	3.92s	287.27s
8_1	12	2387	2.74s	7458	16.23s	3.53s	2530	3.58s	1.71s
8_2	12	2387	2.76s	7458	16.16s	1.11s	2530	3.61s	1.11s
8_3	12	2387	2.70s	7458	16.10s	3.42s	2530	3.66s	22.32s
8_4	12	2387	2.73s	7458	16.22s	1.22s	2530	3.64s	1.38s
8_5	12	2387	2.70s	7458	16.07s	1.74s	2530	3.63s	>20min
9_0	13	3008	3.92s	9547	25.63s	2.87s	3211	5.46s	5.28s
9_1	13	3008	3.90s	9547	25.62s	2.35s	3211	5.45s	2.55s
9_2	13	3008	3.94s	9547	25.61s	6.42s	3211	5.47s	2.29s
9_3	13	3008	3.88s	9547	25.69s	14.01s	3211	5.35s	170.71s
9_4	13	3008	3.90s	9547	25.40s	1.63s	3211	5.42s	588.70s
9_5	13	3008	3.93s	9547	25.76s	20.88s	3211	5.49s	21.04s
10_0	14	3701	5.72s	11894	39.28s	16.21s	3974	8.02s	1.83s
10_1	14	3701	5.73s	11894	38.74s	14.25s	3974	8.02s	660.17s
10_2	14	3701	5.67s	11894	39.43s	7.88s	3974	8.00s	8.90s
10_3	14	3701	5.68s	11894	39.07s	1.45s	3974	7.96s	5.50s
10_4	14	3701	5.67s	11894	39.44s	11.82s	3974	7.95s	7.52s
10_5	14	3701	5.65s	11894	39.31s	1.76s	3974	8.01s	>20min

Rules (CHR) is less powerful in the sense that CHR rules can only rewrite items at the top-level conjunction. CHR implementations are also not deeply integrated with high-level modelling languages in the way Cadmium and Zinc are.

The Conjure system [16] for automatic type refinement accepts models in the high-level constraint specification language ESSENCE and transforms them into models in a sublanguage, ESSENCE', roughly corresponding to a Zinc-to-MiniZinc translation. Conjure's focus is on automatic modelling: the generation of a family of correct but less abstract models that a given input model gives rise to. Our current goal with Cadmium somewhat differently is to have a convenient, all-purpose, highly flexible 'plug-and-play' model rewriting platform.

We have only really begun to explore the possibilities of linearisation of MiniZinc models using Cadmium. There are other decompositions based on Boolean variables $[\![x \leqslant d]\!]$ which could be explored; see e.g. [17, 18]. There are many relaxations and combinations to explore. We can investigate how many IP modelling "tricks" can be implemented using concise Cadmium analysis and rewriting.

On the technical side, we believe data-independent model transformation is a promising direction to take. It would for example mean to postpone unfolding comprehensions, and to transform according to the *derived* rather than present kind of an expression (i.e. constant vs. variable at solve time). We would expect transformation efficiency to greatly improve in this way.

Acknowledgements. This work has taken place with the support of the members of the G12 project.

References

1. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: LOPSTR'03. (2003) 214–232
2. Frisch, A.M., Grum, M., Jefferson, C., Hernandez, B.M., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In: IJCAI'07. (2007)
3. Garcia de la Banda, M.J., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. [19] 700–705
4. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. [20] 529–543
5. Duck, G.J., Stuckey, P.J., Brand, S.: ACD term rewriting. In Etalle, S., Truszczynski, M., eds.: ICLP'06. Volume 4079 of LNCS., Springer (2006) 117–131
6. McKinnon, K., Williams, H.: Constructing integer programming models by the predicate calculus. Annals of Operations Research **21** (1989) 227–246
7. Refalo, P.: Linear formulation of constraint programming models and hybrid solvers. In Dechter, R., ed.: 6th Intl. Conf. on Principles and Practice of Constraint Programming (CP'00). Volume 1894 of LNCS., Springer (2000) 369–383
8. Rafeh, R., Garcia de la Banda, M.J., Marriott, K., Wallace, M.: From Zinc to design model. In Hanus, M., ed.: 9th Intl. Symposium on Practical Aspects of Declarative Languages (PADL'07). Volume 4354 of LNCS., Springer (2007) 215–229
9. Frühwirth, T.: Theory and practice of Constraint Handling Rules. Journal of Logic Programming **37**(1-3) (1998) 95–138
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 system. In Nieuwenhuis, R., ed.: Rewriting Techniques and Applications (RTA'03). Volume 2706 of LNCS., Springer (2003) 76–87
11. Li, Q., Guo, Y., Ida, T.: Modelling integer programming with logic: Language and implementation. IEICE Transactions of Fundamentals of Electronics, Communications and Computer Sciences **E83-A**(8) (2000) 1673–1680
12. Balas, E.: Disjunctive programming: Properties of the convex hull of feasible points. Discrete Applied Mathematics **89**(1-3) (1998) 3–44
13. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic. (1968) 115–125 Reprinted in J. Siekmann and G. Wrightson (eds.), Automation of Reasoning, vol. 2, pp. 466-483, Springer, 1983.
14. Hooker, J.: Integrated Methods for Optimization. Springer (2007)
15. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. Volume 4510 of LNCS., Springer (2007) 1–15
16. Frisch, A.M., Jefferson, C., Hernández, B.M., Miguel, I.: The rules of constraint modelling. In Kaelbling, L.P., Saffiotti, A., eds.: 19th International Joint Conference on Artificial Intelligence (IJCAI'05). (2005) 109–116
17. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. [19] 590–603
18. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation = lazy clause generation. [20] 544–558
19. Benhamou, F., ed.: 12th International Conference on Principles and Practice of Constraint Programming (CP'06). Volume 4204 of LNCS., Springer (2006)
20. Bessière, C., ed.: 13th International Conference on Principles and Practice of Constraint Programming (CP'07). Volume 4741 of LNCS., Springer (2007)

A Resulting LP format

The following is the result of applying the MiniZinc-to-LP format transformation (using the *Big-M* linearisation of `all_different`) to the Golomb Ruler problem of Section 2.1:

```
Minimize mark{3}
Subject To
   mark{0} = 0
   mark{1} >= 1
   mark{2} - 1 differences{3} - 1 mark{1} = 0
   mark{3} - 1 differences{4} - 1 mark{1} = 0
   mark{3} - 1 differences{5} - 1 mark{2} = 0
   differences{0} - mark{1} = 0
   differences{1} - mark{2} = 0
   differences{2} - mark{3} = 0
   mark{1} + mark{2} - 1 mark{3} <= -1
   mark{1} - 1 mark{2} <= -1
   differences{5} + 17 V_107 - 1 differences{4} <= 16
   -1 V_108 - 1 V_107 <= -1
   differences{4} + 17 V_108 - 1 differences{5} <= 16
   differences{5} + 17 V_105 - 1 differences{3} <= 16
   -1 V_106 - 1 V_105 <= -1
   differences{3} + 17 V_106 - 1 differences{5} <= 16
   differences{4} + 17 V_103 - 1 differences{3} <= 16
   -1 V_104 - 1 V_103 <= -1
   differences{3} + 17 V_104 - 1 differences{4} <= 16
   differences{5} + 17 V_101 - 1 differences{2} <= 16
   -1 V_102 - 1 V_101 <= -1
   differences{2} + 17 V_102 - 1 differences{5} <= 16
   differences{4} + 17 V_99 - 1 differences{2} <= 16
   -1 V_100 - 1 V_99 <= -1
   differences{2} + 17 V_100 - 1 differences{4} <= 16
   differences{3} + 17 V_97 - 1 differences{2} <= 16
   -1 V_98 - 1 V_97 <= -1
   differences{2} + 17 V_98 - 1 differences{3} <= 16
   differences{5} + 17 V_95 - 1 differences{1} <= 16
   -1 V_96 - 1 V_95 <= -1
   differences{1} + 17 V_96 - 1 differences{5} <= 16
   differences{4} + 17 V_93 - 1 differences{1} <= 16
   -1 V_94 - 1 V_93 <= -1
   differences{1} + 17 V_94 - 1 differences{4} <= 16
   differences{3} + 17 V_91 - 1 differences{1} <= 16
   -1 V_92 - 1 V_91 <= -1
   differences{1} + 17 V_92 - 1 differences{3} <= 16
   differences{2} + 17 V_89 - 1 differences{1} <= 16
   -1 V_90 - 1 V_89 <= -1
   differences{1} + 17 V_90 - 1 differences{2} <= 16
   differences{5} + 17 V_87 - 1 differences{0} <= 16
   -1 V_88 - 1 V_87 <= -1
   differences{0} + 17 V_88 - 1 differences{5} <= 16
   differences{4} + 17 V_85 - 1 differences{0} <= 16
   -1 V_86 - 1 V_85 <= -1
   differences{0} + 17 V_86 - 1 differences{4} <= 16
   differences{3} + 17 V_83 - 1 differences{0} <= 16
   -1 V_84 - 1 V_83 <= -1
   differences{0} + 17 V_84 - 1 differences{3} <= 16
   differences{2} + 17 V_81 - 1 differences{0} <= 16
   -1 V_82 - 1 V_81 <= -1
   differences{0} + 17 V_82 - 1 differences{2} <= 16
   differences{1} + 17 V_79 - 1 differences{0} <= 16
   -1 V_80 - 1 V_79 <= -1
   differences{0} + 17 V_80 - 1 differences{1} <= 16
Bounds
   0 <= mark{0} <= 16
   0 <= mark{1} <= 16
   0 <= mark{2} <= 16
   0 <= mark{3} <= 16
   0 <= differences{0} <= 16
   0 <= differences{1} <= 16
   0 <= differences{2} <= 16
   0 <= differences{3} <= 16
   0 <= differences{4} <= 16
   0 <= differences{5} <= 16
   0 <= V_99 <= 1
   0 <= V_97 <= 1
   0 <= V_98 <= 1
   0 <= V_95 <= 1
   0 <= V_96 <= 1
   0 <= V_93 <= 1
   0 <= V_94 <= 1
   0 <= V_91 <= 1
   0 <= V_92 <= 1
   0 <= V_89 <= 1
   0 <= V_90 <= 1
   0 <= V_87 <= 1
   0 <= V_88 <= 1
   0 <= V_85 <= 1
   0 <= V_86 <= 1
   0 <= V_83 <= 1
   0 <= V_84 <= 1
   0 <= V_81 <= 1
   0 <= V_82 <= 1
   0 <= V_79 <= 1
   0 <= V_80 <= 1
   0 <= V_107 <= 1
   0 <= V_108 <= 1
   0 <= V_105 <= 1
   0 <= V_106 <= 1
   0 <= V_103 <= 1
   0 <= V_104 <= 1
   0 <= V_101 <= 1
   0 <= V_102 <= 1
   0 <= V_100 <= 1
General
   mark{0}
   mark{1}
   mark{2}
   mark{3}
   differences{0}
   differences{1}
   differences{2}
   differences{3}
   differences{4}
   differences{5}
   V_80
   V_79
   V_82
   V_81
   V_84
   V_83
   V_86
   V_85
   V_88
   V_87
   V_90
   V_89
   V_92
   V_91
   V_94
   V_93
   V_96
   V_95
   V_98
   V_97
   V_99
   V_100
   V_102
   V_101
   V_104
   V_103
   V_106
   V_105
   V_108
   V_107
End
```