A Rule-based System for Model Transformation

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

NICTA, Victoria Research Lab, University of Melbourne, Australia

Abstract. We describe the constraint model transformation system Cadmium, focussing on its application. We discuss how we use it to reduce solver-independent models in the medium-level modelling language MiniZinc into equivalent models for finite domain constraint solvers and integer linear programming solvers.

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 and without having to make subordinate modelling decisions or even to commit to a particular solving approach. Examples for decisions that depend on the target solver are the representation of variables of a complex type or the translation of constraints into those provided. Such decisions need to be taken if concrete solvers such as ILOG Solver or Eclipse are used directly.

The problem solving process is thus broken into, first, the development of a high-level, solver-independent, conceptual model and, second, the mapping of it into 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.

In terms of system support, there is an imbalance between the tasks of model formulation, for which well-designed, open high-level languages exists, and model transformation, which is typically done by fixed procedures inaccessible to the modeller. It is hard to see, however, that there is a single best set of transformations that can be wrapped and packed away. We therefore believe that a strong requirement on a model transformation process and platform is *flexibility*.

In this paper we describe our model transformation system Cadmium [4] with a focus on its use. Cadmium, in contrast to comparable systems, is term-rewriting based. 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, in our view, transformations are often very compact and comprehensible.

We have found that Cadmium addresses the flexibility requirement well, and we provide evidence for this statement in this paper. We report on several transformations, including two major ones over models in the medium-level

modelling language MiniZinc [5], a subset of Zinc. One transformation maps MiniZinc models into a form easily executable by ordinary Finite Domain constraint solvers. The second transformation generates linear constraints solvable by (Integer) Linear Programming solvers.

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 [5] is a subset of Zinc closer to existing CP languages and that is still suitable as a medium-level constraint modelling language. FlatZinc, also described in [5], is a low-level subset of Zinc. 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 = 5;
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;

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

    % Symmetry breaking
constraint mark[2] - mark[1] < mark[m] - mark[m-1];

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 variable arrays `mark` and `differences`. The elements of either array are variables taking 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. More constraint items follow, specifying the properties of the desired Golomb rulers.
– 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 ZINC and its sublanguages is available in [3, 6, 5].

2.2 The CADMIUM model transformation system

CADMIUM [4] 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 can be transformed into another by a CADMIUM program (or 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.

Transforming ZINC models with CADMIUM is very natural because of the tight representational integration between the two languages. In technical terms, our integrated ZINC solver tool reads a ZINC model and a CADMIUM transformation specification according to which the model is transformed. The resulting model can be output for inspection or further use, or it can be immediately solved by the built-in FD constraint solver.

A CADMIUM program is a sequence of CADMIUM rules of the form

Head $\Leftrightarrow$ *Body*

where *Head* and *Body* are arbitrary terms that in particular can be or contain ZINC expressions. Any expression from the input model matching *Head* is rewritten to the expression *Body*. The rules in the program are repeatedly applied until no more applications are possible. The obtained model is the result of the transformation.

To emphasise the free combination of ZINC expressions with arbitrary term constructors, we continue marking ZINC keywords as in **constraint**, **int**, **array**.

Example 1. The following is a one-rule CADMIUM transformation:

```
X + 0 ⇔ X
```

Given this program, CADMIUM will transform the ZINC item

```
constraint a + 0 = 10;
```

into the simplified form

```
constraint a = 10;
```

□

CADMIUM has several features that make implementing model transformation easier:

– associative commutative matching;
– distributive or conjunctive context matching;
– user-definable guards on rules;
– staged transformations.

We describe them in the following.

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. $+$, $*$, $\wedge$, $\vee$, $\cup$, $\cap$. In CADMIUM, the order and nested structure of expressions constructed form AC operators does not matter. Because of AC matching, the number of rules required to express a transformation can be significantly reduced.

Example 2. Consider the rule from Example 7 and the ZINC item:

```
constraint 0 + a = 10;
```

Thanks to AC matching, the rule is still applicable to the expression `0 + a`, despite the order of the arguments for `+` being reversed. □

Conjunctive context matching. CADMIUM supports conjunctive context (CC) matching, which makes it possible for rules to match against non-local information. At the core of CC matching is the following observation: given a conjunction $X \wedge Y$, fact X is true in Y as well as in all subterms of Y. We say that X is in the *conjunctive context* of Y. Likewise, Y is in the conjunctive context of X.

Example 3. Consider the expression

```
(a < b) ∧ f(1 + bool2int(b > a))
```

The expression `a < b` is in the conjunctive context of the sub-term `b > a`. We can therefore deduce that `b > a` must be false, and simplify the expression:

```
(a < b) ∧ f(1 + bool2int(false))
```

□

A CADMIUM rule of the form:

CCHead $\setminus$ *Head* $\Leftrightarrow$ *Body*

uses CC matching. The expression matching *Head* will be rewritten to *Body* provided an expression matching *CCHead* is in the conjunctive context.

Example 4. Reconsider the transformation from Example 3. It can be achieved by the following CADMIUM rule:

```
X < Y \ X > Y ⇔ false
```

□

Example 5. CC matching can be used to implement parameter substitution in constraint models as follows:

```
X = C \ X ⇔ 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)) ∧ a = 3
```

After exhaustively applying the rule, the result is:

```
f(3,3+b,g(3)) ∧ a = 3
```

□

A more detailed explanation of conjunctive context can be found in [4].

User-definable guards. CADMIUM supports rule with guards. The syntax is:

CCHead \ *Head* ⇔ *Guard* | *Body*

Such a rule can only be applied if the *Guard* holds, that is, if it can be rewritten to `true`. CADMIUM provides a number of simple guards, e.g. `is_int(X)`, `is_float(X)`. Guards can also be defined by the user via rules.

Example 6. The following program is the same as that in Example 5, except rule application has been restricted by guards.

```
X = C \ X ⇔ is_var(X) ∧ is_num(C) | C

is_num(C) ⇔ is_int(X) | true
is_num(C) ⇔ is_float(X) | true
```

The guards `is_int`, `is_float`, and `is_var` are built-in guards, and `is_num` is a user-defined guard. The guard `is_num(C)` succeeds if `C` is an integer or a floating point number. □

Staged transformations. For complex mappings, CADMIUM supports *staged* transformations. So far, all of the transformations have been simply sets of rules. A staged transformation consists of several rule sets run in sequence. Staged transformations are declared via `transform` declarations. For example,

```
transform main = stage_1, stage_2;
```

defines a transformation `main` that consists of the sub-transformations `stage_1` and `stage_2`. The output of `stage_1` becomes the input to `stage_2`, and the subsequent output of `stage_2` is the final result of `main`.

The simplest transformations are sets of rules. Rules sets are given names via `ruleset` declarations, e.g.

```
ruleset simplify_ints;
    X + 0 ⇔ X;
    X * 0 ⇔ 0;
```

Rule sets can import other rule sets, e.g.

```
ruleset simplify_num;
    import simplify_ints;
    import simplify_floats;
```

Thus, `simplify_num` is in effect the union of the two rulesets `simplify_ints` and `simplify_floats`. Importing of rulesets can also be inlined inside transformation declarations; e.g.

```
transform main = (simplify_ints | simplify_floats), stage_2;
```

is equivalent to

```
transform main = simplify_num, stage_2;
```

2.3 Representation of ZINC models in CADMIUM

The CADMIUM rewriting engine operates on terms, similar to terms in Prolog. This means that a ZINC model must be represented as a CADMIUM term in a standard way. We do so as follows:

- Each ZINC item is given some term representation, e.g. **constraint** `X+3` is represented as `'!constraint'(X+3)`. Note that the '!' is added to functor '`constraint`' to help avoid name space clashes.
- All ZINC items in the model are joined by conjunction, thus

  constraint `X = 3;`
  solve satisfy`;`

  is represented as

  `'!constraint'(X = 3)` $\wedge$ `'!solve'('!satisfy')`

  Therefore, ZINC items can be looked up by rules using conjunctive context.
- The conjunction of ZINC items is wrapped by a top-level `model` functor. This means that it is possible to rewrite the entire model at once:

  model `Model` $\Leftrightarrow$ `...`

  This representation allows top-down model transformation in the way non-term-rewriting-based approaches work. However, in our experience, top-down transformations are rarely needed.

Much low-level details about model representation is hidden from the user, since the CADMIUM parser accepts a ZINC-like syntax. However, generalising the ZINC grammar for CADMIUM has led to ambiguity. For example, consider `T: X` in the head of a rule. It can match many different things in ZINC, e.g. a variable declaration, an array index-value pair, or a record identifier-value pair. Some ambiguity is currently resolved by additional keywords. Furthermore, CADMIUM has shadow versions of many ZINC operators which are evaluated by the CADMIUM engine. For instance, `1+2` represents a ZINC addition whereas `1!+2` is evaluated to `3`.

A good solution to the issues raised by merging the base grammar of CADMIUM with ZINC's grammar needs to achieve both rigour and conciseness, and is on-going work.

3 Model transformations

For the reader to gain an understanding of the conciseness and readability possible in CADMIUM, we give some representative example rules in the following sections. We emphasise that most rules are copied verbatim from our actual source code and (except for minor stylistic improvements) are thus executable. Of others we use generalised versions.

3.1 Elimination of enumerated types

As a gentle introduction to transformations, we start with a simple task: the elimination of enumerated types. Enumerated types provide a set of named alternatives in Zinc. Here is an example definition:

```
enum animals = {cat, dog, unicorn};
```

Typically, target solvers do not support enumerated types, so the type constants need to be represented as integer parameters. The following transformation applies a standard mapping to a model, replacing the ith enumerated type constant by the integer i.

```
enum T = Symbols ⇔
    (type T = 1..set_size(Symbols)) ∧
    refine_enum_type(T, Symbols, 1)

refine_enum_type(_, {}, _) ⇔ true

refine_enum_type(T, {S ! Symbols}, I) ⇔
    T: S = I ∧ refine_enum_type(T, Symbols, I !+ 1)
```

The function `set_size` used here is handled by (the obvious) rules in the Cadmium standard library. Given the enumerated type definition above, the transformation will generate:

```
type animals = 1..3;
animals: cat = 1;
animals: dog = 2;
animals: unicorn = 3;
```

A fixed, linear mapping of symbols to integers is applied here. It should be easy to see that it is not hard to write rules using instead a mapping chosen by the modeller, which may sometimes be desirable.

3.2 Type refinement

As a second small example let us consider a simple case of type refinement: the reduction of a set type to a Boolean array type. The array index set is the universe of the original set.

```
var set of T: S ⇔ array[T] of var bool: A ∧ refine(S, A)

refine(S, A) \ P subset S ⇔ forall([ A[E] | E in P ])
```

The first rule replaces a set variable declaration by a corresponding array variable declaration, using a fresh identifier, and installs a refinement token at the model top level, i.e. in the conjunctive context of the entire model. By the second rule, subset constraints fitting the refinement token are replaced by the appropriate constraint on the new array variable. Similar rules for all other set constraints would also be part of this rule set. In a concluding stage, the refinement token is deleted.

This two-stage transformation rewrites the model

```
var set of 1..5: T;
constraint B → (1..3 subset T);
var bool: B;
```

into the refined model

```
array[1..5] of var bool: V1;
constraint B → forall([ V1[V2] | V2 in 1..3 ]);
var bool: B;
```

The new identifiers `V1`, `V2` are freshly introduced by the transformation and uniquely named by the Cadmium engine.

Note that the rules did not specify the set element type `T`. Zinc allows deeply nested types, and further refinement rules may be needed to reduce the so generated array indexed by `T` into e.g. an integer-indexed array.

It is clear that this proof-of-concept transformation is still a considerable distance from a comprehensive type refinement system as present in Conjure [7]. We hope to be able to formulate an equivalent system using concise Cadmium transformations.

3.3 Transforming MiniZinc to FlatZinc

Next, we present elements of the Cadmium variant of the MiniZinc-to-FlatZinc conversion described in [5].

1. Model normalisation.
   Zinc allows much choice 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 unification of synonyms:
   `(constraint C) ∧ (constraint D) ⇔ constraint C ∧ D`
   `X == Y ⇔ X = Y`
2. Predicate inlining.
   We 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 significant case of a top-down transformation.
3. The next steps, while defined separately and listed in sequence, depend on and enable one another. In a non-term-rewriting approach, an iteration algorithm would be needed to compute the mutual fixpoint. In Cadmium, each individual 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.
   (a) Partial evaluation.
   We here apply rules that simplify the model by taking into account the semantics of Zinc constructs. Consider:

```
X ∨ true ⇔ true
X + Y ⇔ is_int(X) ∧ is_int(Y) | X !+ Y
L..U:X \ X ⩽ C ⇔ is_int(U)∧ is_int(C) ∧ U!⩽C | true
```

(b) Compound built-in unfolding.
Expressions with e.g. **sum**, **forall** are compound expressions in Zinc. They are inlined by rules such these:

```
sum([]) ⇔ 0
sum([E ! Es]) ⇔ E + sum(Es)
```

(c) Parameter substitution.
We use the conjunctive context of an expression:

```
T: X = E \ X ⇔ is_int(E) | E;
```

(d) Comprehension unfolding.

```
[E | X in L..U] ⇔ L !> U | []
[E | X in L..U] ⇔ [subst(X=L, E) ! [E | X in L+1..U]]
```

4. Boolean/numeric/set normalisation and decomposition.
This step again consists of separate sets of independent, composed transformations; one set of rule sets each for normalising and for decomposing constraints involving the respective types.
Numeric normalisation, for example, joins constants on opposite sides of an inequality:

```
C+E < D ⇔ is_int(C) ∧ is_int(D) | E < D!-C
```

Our choice for Boolean normalisation is to establish a negation normal form by rules such as

```
not (X → Y) ⇔ X ∧ not Y
not (A ⩾ B) ⇔ A < B
```

Boolean decomposition.
In a first step, we extract complex subexpressions using a **let** formulation:

```
C → D ⇔ is_not(is_variable(C)) |
    let { var bool: B = C } in (B → D)
```

Guard `is_not(`G`)` holds if G rewrites to **false**. The inner guard `is_variable(`X`)` holds if X is a model variable, which includes Cadmium identifiers (tested by `is_var`) and lookups to variable arrays, e.g. `a[0]` is a Zinc variable if `a` is an array of variables.

Numeric decomposition.
The rules here are slightly more complex than in the Boolean case:

```
A[X] ⇔ is_not(is_variable(X)) ∧ is_not(is_int(X)) |
    let { var lbound(X)..ubound(X): W = X } in A[W]

max(X,Y) ⇔ is_not(is_variable(X)) ∧ is_not(is_int(X)) |
    let { var lbound(X)..ubound(X): W = X } in max(W, Y)
```

The `*bound` functors are used to derive a tight type for the new variable. They are defined using rules such as the following:

```
lbound(C) ⇔ is_int(C) | C
L..U: X \ lbound(X) ⇔ L
lbound(X + Y) ⇔ term(X) | lbound(X) + lbound(Y)
```

The local variable declarations introduced during decomposition are lifted to the model top level by a command provided by the CADMIUM engine:

```
let { var T: X } in E ⇔ top_level(E, T: X)
```

5. FLATZINC format.
   The final step is to turn the constraints, now decomposed and in normal form, into their FLATZINC form:

```
Z = max(X, Y) ⇔ int_max(X, Y, Z)
X = A[Y] ⇔ is_not(is_int(Y)) | array_int_element(Y,A,X)
```

Example 7. Here are some of the constraints resulting in the MINIZINC-to-FLATZINC conversion of the Golomb Ruler problem, Section 2.1:

```
constraint int_plus(differences[3], mark[1], mark[2]);
constraint int_lt(mark[1], mark[2]);
constraint int_eq(mark[0], 0);
constraint all_different(differences);
```

□

3.4 Transforming MINIZINC into linear (Mixed Integer Programming) format

There are many similarities between the MINIZINC-to-FLATZINC and this transformation which generates a model of linear constraints only. In principle, it would suffice to transform FLATZINC to a MIP format; however, the generic Boolean decomposition is inappropriate for this purpose. So we just share many of the subtransformations. More specifically, rule sets 1–3 are identical. Rule sets 4–9 are detailed below.

The transformation is based on the work by McKinnon and Williams [8] and Li et al. [9]. We simplified the transformation and made some steps, such as Boolean normalisation, more explicit. Li et al. define the modelling language $\mathcal{L}^+$ consisting of linear arithmetic constraints, Boolean operators, and some further formulas such as at_most or at_least. Steps of the transformation described in [9]:

- Transformation of $\mathcal{L}^+$ into negation normal form.
- Transformation of simplified $\mathcal{L}^+$-formulas into Γ-formulas, which are of the form $\Gamma_m\{P_1, \ldots, P_n\}$ where each P_i is a Γ-formula or constraint, and $\Gamma_m\{P_1, \ldots, P_n\}$ means at least m formulas of $\{P_1, \ldots, P_n\}$ are true.
- Elimination of negated Γ-formulas.
- Flattening of nested Γ-formulas.
- Transformation of Γ-formulas into MIP-expressions.

Our transformations also use the previously defined Γ-formulas. After applying the previously defined rulesets 1–3, we do further normalisation and decomposition. We then generate Γ-formulas which are further transformed into a linear form of MINIZINC. As a final optional step, we can also transform these linear models into CPLEX LP format for directly feeding them into most of the currently available MIP solvers.

4. Boolean/numeric normalisation and decomposition.
   In addition to the previously defined normalisations and decompositions, we decompose different generic constraints such as the domain constraint:

   ```
   X in A..B ⇔ is_int(A) ∧ is_int(B) | A ⩽ X ∧ X ⩽ B
   X in S ⇔ is_set(S) | exists(map('='(X), set2list(S)))
   ```

   For mapping domain constraints we discern two cases: ranges which can simply be mapped onto two inequalities, and the others where the domain is a general set of values. We map this second case to a disjunction over the values of the domain (`map(F,L)` applies `F` to the elements of `L`).

   Element constraint:

   ```
   Y = A[X] ⇔
       is_variable(X) | exists(map(ec_decomp(A, X, Y),
           range2list(lbound(X), ubound(X))))

   ec_decomp(A, X, Y, I) ⇔ A[I] = Y ∧ X = I
   ```

   The element constraint is transformed into a disjunction over all possible values of `X`; each of the disjuncts fixes `Y` according to a given `X`.

   Alldifferent constraint:

   ```
   all_different(X) ⇔
       forall(I,J in index_set(X) where I < J) (X[I] != X[J])
   ```

   We simply decompose the alldifferent constraint into a conjunction of inequations between all variable pairs.
   Furthermore strict inequalities, inequations, minimum and maximum are transformed into a linear form. The above rules are the most straightforward decompositions of element and alldifferent. More refined decompositions are described in [10].
5. Negation normal form (established by Boolean normalisation).
   As in the MiniZinc-to-FlatZinc mapping, we transform the formulas into negation normal form. An example:

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

   is transformed into:

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

6. Rewriting binary into n-ary conjunctions/disjunctions (`conj()`, `disj()`) so that they can be further transformed into Γ-formulas.

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

   The formula from the example above:

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

   becomes:

   ```
   gamma([gamma([x - y ⩽ 5, x - y ⩾ 5], 2, 1),
          z ⩾ 1], 2, 1)
   ```

7. Boolean to linear transformations.
   These are mainly based on the Γ to linear transformations, but are simpler, since all constraints were previously normalised.

```
constraint gamma(Cs, N, N) ⇔ constraint forall(Cs)
constraint gamma(Cs, N, M) ⇔
    N !> M | constraint true → gamma(Cs, N, M)
constraint gamma(Cs, N, M) ∧ Ds ⇔
    N !> M | constraint true → gamma(Cs, N, M) ∧ Ds;

B → gamma(Cs, _, M) ⇔ g_aux(B, Cs, M, [])

g_aux(B0, [C ! Cs], M, Bs) ⇔
    let { var bool: B } in
    ( (B → C) ∧ g_aux(B0, Cs, M, [bool2int(B) ! Bs]))
g_aux(B0, [], M, Bs) ⇔ B0 → sum(Bs) ⩾ M

B → E ⩽ F ⇔ E-F ⩽ ubound(E-F) * (1-bool2int(B))
```

The first three rules transform top-level Γ-formulas. The fourth and fifth rule transform a formula $B \rightarrow \Gamma_m(Cs)$ into a conjunction of rules $B_i \rightarrow C_i$. The B_i are accumulated in a list. The sixth rule then adds $B_0 \rightarrow \sum(B_i) \geqslant m$. Finally the last rule rewrites $B_0 \rightarrow \sum(B_i) \geqslant m$ into the form $\sum(B_i) \geqslant B_0 m$.

We optimise the transformation by distinguishing some special cases:

```
constraint
    gamma_aux(B0, [B ! Cs], M, Bs) ⇔
    is_bool_var(B) | g_aux(B0, Cs, M, [bool2int(B) ! Bs])
```

Looking at part of our example and assuming x and y are in 0..10:

```
gamma([x - y ⩽ 5, x - y ⩾ 5], 2, 1)
```

is stepwise transformed as follows (ignoring **bool2int** for brevity):

```
B → gamma([x - y ⩽ 5, x - y ⩾ 5], 2, 1)
g_aux(B,[x - y ⩽ 5, x - y ⩾ 5], 2, 1)
B1 → x - y ⩽ 5 ∧ B2 → x - y ⩾ 5 ∧ B → B1 + B2 ⩾ 1
x - y - 5 ⩽ 5 *(1 - B1) ∧ 5 - x + y ⩽ 15 *(1 - B2) ∧
      1 - B1 - B2 ⩽ 1 - B
```

8. Boolean-to-integer.
   In this ruleset we transform Boolean variables into 0-1 integer variables by directly substituting the type (and reusing the variables).
   ```
   bool ⇔ 0..1
   bool2int(B) ⇔ B
   ```
9. A possible concluding stage of the linearisation prints out the linear model in CPLEX LP format using CADMIUM's I/O facilities.

Example 8. The following is part of the result of applying the MINIZINC-to-LP format transformation to the Golomb Ruler problem from Section 2.1:

```
Minimize mark{3}
Subject To
    mark{0} = 0
    mark{2} - 1 mark{1} - 1 differences{3} = 0
    mark{1} - 1 mark{2} <= -1
    -1 V_84 - 1 V_85 <= -1
    differences{1} + 17 V_85 - 1 differences{0} <= 16
```

```
    differences{0} + 17 V_84 - 1 differences{1} <= 16
    ...
Bounds
    0 <= differences{0} <= 16
    0 <= mark{1} <= 16
    0 <= V_84 <= 1
    0 <= V_85 <= 1
    ...
General
    mark1
    differences0
    V_84
    V_85
    ...
```

4 Case studies

We tested and evaluated the MiniZinc to FlatZinc as well as MiniZinc to LP format transformations on several different models:

- alpha: Gecode alpha example
- eq20: Solving twenty linear constraints
- golomb: Golomb rulers ($m \in \{4, 6, 8, 10\}$)
- jobshop: Square job scheduling (2x2, 4x4, 6x6, 8x8)
- mdknapsack: Multidimensional knapsack problem ($n = 5, m = 3$ and $n = 100, m = 5$)
- packing: Packing squares into a rectangle (4)
- perfsq: Find a set of integers the sum of whose squares is itself a square (with maximum integer 10, 20, or 30)
- n-queens: 8, 10, 20
- radiation: Radiation treatment planning (4x4, 5x5)
- warehouses: Warehouse construction problem

The experiments where performed on a 3.4Ghz Intel Pentium D with 4Gb RAM computer running Linux. The Flatzinc models were solved by the G12 finite domain solver, the LP models were solved using CPLEX 10.0. The solvers were aborted if they did not return a result within 5 minutes.

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 LP solver means that the LP translations are often better overall. Some of the slightly bigger examples (golomb8 and golomb10, jobshop8x8, mdknapsack2, perfsq30, 20-queens, and radiation5) show that translations times do scale, but the solve times can increase dramatically. For some examples we can see a clear advantage for the FD solver (n-queens, golomb, jobshop), whereas for other examples the MIP solver performs better(mdknapsack, perfsq, radiation). Most of the other examples are small, where the translation time often dominates on solving time.

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

name	mzn	fzn			lp format		
	lines	lines	trans.[s]	solve[s]	lines	trans.[s]	solve[s]
alpha	52	53	0.29	0.17	2356	1.52	0.09
eq20	63	82	0.14	0.09	43	0.19	0.00
golomb4	11	16	0.14	0.04	146	0.19	0.00
golomb6	11	27	0.15	0.09	809	0.32	0.059
golomb8	11	42	0.16	0.71	2765	0.95	11.54
golomb10	11	61	0.39	229.24	7106	6.87	-
jobshop2x2	20	18	0.13	0.05	37	0.17	0.00
jobshop4x4	22	141	0.16	0.09	227	0.21	0.01
jobshop6x6	24	492	0.26	0.55	749	0.33	0.67
jobshop8x8	26	1191	0.95	-	1771	1.32	-
mdknapsack1	21	16	0.13	0.04	25	0.18	0.00
mdknapsack2	75	175	0.39	-	217	0.64	0.31
packing	32	237	0.15	0.09	378	0.22	0.00
perfsq10	16	89	0.14	0.08	948	0.36	0.04
perfsq20	16	161	0.14	0.79	3068	1.27	0.32
perfsq30	16	233	0.15	55.31	6388	3.63	3.17
8-queens	9	86	0.14	0.09	613	0.26	0.01
10-queens	9	86	0.14	0.09	613	0.26	0.02
20-queens	9	572	0.26	0.11	4039	1.38	-
radiation4	40	553	0.32	1.12	1689	0.83	0.02
radiation5	41	1032	0.56	-	3084	2.07	0.13
warehouses	45	476	0.22	1.21	1495	0.50	0.77

5 Final remarks

Cadmium is one of only a few purpose-built systems targetting constraint model transformation, and among these, has particular strengths. Constraint Handling 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 [7] 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 platfom.

As a case in point for such a platform (beside the content of this paper), consider the recent work on preprocessing constraint models for stochastic local search [11]. It deals with an architecture in which models in a language comparable to MiniZinc can be simplified. The studied simplifications include a translation into negation normal form, simple consistency reasoning on the variable domains, substitution of a variable occurrence if its conjunctive context

determines its value ('dealiasing'), and forms of partial evaluation. These tasks are already implemented in Cadmium rules or appear to be easily doable. Also pointed out in [11] is the fact that, since subtransformations can interact, their mutual fixpoint should be computed. This fixpoint computation comes for free using a term-rewriting-based approach with merged rule sets as in Cadmium.

Acknowledgements. This work has taken place with the support of the members of the G12 project. We also thank the reviewers for their comments.

References

1. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Proc. of LOPSTR 2003. (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: Proceedings of IJCAI-07. (2007)
3. de la Banda, M.J.G., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. In Benhamou, F., ed.: Proc. of 12th International Conference on Principles and Practice of Constraint Programming (CP'06). Volume 4204 of LNCS., Springer (2006) 700–705
4. Duck, G.J., Stuckey, P.J., Brand, S.: ACD term rewriting. In Etalle, S., Truszczynski, M., eds.: ICLP. Volume 4079 of Lecture Notes in Computer Science., Springer (2006) 117–131
5. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming (CP'07). (2007) To appear.
6. Rafeh, R., de la Banda, M.J.G., Marriott, K., Wallace, M.: From Zinc to design model. In Hanus, M., ed.: Proc. 9th International Symposium of Practical Aspects of Declarative Languages (PADL'07). Volume 4354 of LNCS., Springer (2007) 215–229
7. 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
8. McKinnon, K., Williams, H.: Constructing integer programming models by the predicate calculus. Annals of Operations Research **21** (1989) 227–246
9. 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
10. Refalo, P.: Linear formulation of constraint programming models and hybrid solvers. In Dechter, R., ed.: Proc. of 6th International Conference on Principles and Practice of Constraint Programming (CP'00). Volume 1894 of LNCS., Springer (2000) 369–383
11. Sabato, S., Naveh, Y.: Preprocessing expression-based constraint satisfaction problems for stochastic local search. In Hentenryck, P.V., Wolsey, L., eds.: Proc. of 4th Int. Conf. on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'07). Volume 4510 of LNCS., Springer (2007) 244–259