

Rewriting Constraint Models with Metamodels

Raphaël Chenouard¹ and Laurent Granvilliers¹ and Ricardo Soto^{1,2}

¹Université de Nantes, LINA, CNRS UMR 6241, France

²Escuela de Ingeniería Informática

Pontificia Universidad Católica de Valparaíso, Chile

{raphael.chenouard, laurent.granvilliers, ricardo.soto}@univ-nantes.fr

Abstract

An important challenge in constraint programming is to rewrite constraint models into executable programs calculating the solutions. This phase of constraint processing may require translations between constraint programming languages, transformations of constraint representations, model optimizations, and tuning of solving strategies. In this paper, we introduce a pivot metamodel describing the common features of constraint models including different kinds of constraints, statements like conditionals and loops, and other first-class elements like object classes and predicates. This metamodel is general enough to cope with the constructions of many languages, from object-oriented modeling languages to logic languages, but it is independent from them. The rewriting operations manipulate metamodel instances apart from languages. As a consequence, the rewriting operations apply whatever languages are selected and they are able to manage model semantic information. A bridge is created between the metamodel space and languages using parsing techniques. Tools from the software engineering world can be useful to implement this framework.

Introduction

In constraint programming (CP), users describe properties of problems as constraints involving variables. The computer system calls constraint solvers to calculate the solutions. The automatic mapping from constraint models to solvers is the key issue of this paper. The goal is to develop middle software tools that are able to reformulate and rewrite models according to solving requirements.

Modeling real-world problems requires high-level languages with many constructions such as constraint definitions, programming statements, and modularity features. In the recent past, a variety of languages has been designed for a variety of users and problem categories. On one hand, there are many modeling languages for combinatorial problems such as OPL (Van Hentenryck et al. 1999), Essence (Frisch et al. 2007), and MiniZinc (Nethercote et al. 2007) or numerical constraint and optimization problems such as Numerica (Van Hentenryck, Michel, and Deville 1997) and Realpaver (Granvilliers and Benhamou 2006). On the other hand, constraint solving libraries have

been plugged in computer programming languages, for instance ILOG Solver (Puget 1994), Gecode (Schulte and Tack 2006), and ECLⁱPS^e (Apt and Wallace 2007). In the following, we will only consider modeling languages as input constraint models. However, computer programming languages can be chosen as targets of the mapping process. Our aim is therefore to provide a many-to-many mapping tool that is able to cope with a variety of languages.

Many constructions are shared among the different languages, in particular the definitions of constraints. Other constructions are specific such as classes in object-oriented languages or predicates in logic languages. We propose to embed this collection of concepts in a so-called metamodel, that is a model of constraint models. This pivot metamodel describes the relations between concepts and it encodes in an abstract manner the rules for constraint modeling. This is a considerable improvement of our previous work (Chenouard, Granvilliers, and Soto 2008) which was restricted to a one-to-many mapping approach from a particular modeling language. Moreover, the translations to obtain Flat s-COMMA models were hand-coded and model structures are always flattened like for FlatZinc models (Nethercote et al. 2007). Previous model transformations were also specific to Flat s-COMMA and its structure (e.g. there is no object and no loop to manage). Our pivot metamodel is independent of modeling languages and our approach offers more flexibility in getting efficient executable models.

The rewriting process can be seen as a three-steps procedure. During the first step, the user constraint model is parsed and a metamodel instance is created. During the last step, the resulting program is generated from a metamodel instance. These two steps constitute a bridge between languages — the grammar space — and models — the model space. The middle step may implement rewriting operations over metamodel instances, for instance to transform constraint representations from an integer model to a boolean model. The main interest is to manipulate concepts rather than syntactic constructions. As a consequence, the rewriting operations can be expressed with clarity and they apply whatever languages are chosen.

An interesting work is about the rule-based programming language Cadmium (Duck, Stuckey, and Brand 2006) combining constraint handling rules (Frühwirth 2009) and term rewriting to transform constraint models. The rewriting al-

gorithm matches rules against terms in order to derive some term normal forms. This approach provides a very clear semantics to the mapping procedure and it addresses confluence and termination issues. Considering metamodels allows one to reuse metamodeling tools from software engineering. For instance, ATL (Kurtev, van den Berg, and Jouault 2007) is a general rule-based transformation language mixing model pattern matching and imperative programs, which can be contrasted with term matching in Cadmium. Kermeta (Muller, Fleurey, and Jézéquel 2005) is a transformation framework allowing to handle model elements using object-oriented programs. A benefit of the model-driven approach is to directly manage typed model concepts using the metamodel abstract description.

The remaining of this paper is organized as follows. Section 2 presents the general model-driven transformation framework underlying this work. A motivating example using known CP languages is described in Section 3. The pivot metamodel and rewriting operations are presented in Section 4. Section 5 investigates some transformation experiments on well-known CP models. Finally, Section 6 concludes the paper and details some future work.

Model Engineering Framework

A constraint model is a representation of a problem, written in a language, and having a structure. Our purpose is to transform solver-independent models to solver-dependent models. That may lead

- to change the representation of input models, namely the intrinsic constraint definitions, in order to improve the solving strategy,
- to translate languages, from high-level modeling languages to low-level solver languages or computer programming languages, and
- to modify model structures according to the capabilities of solvers, for instance to make a shift from object-oriented models to logic models based on predicates.

Managing representations supposes to specify constraint transformation rules such as the equivalence of constraint formulations or constraint relaxations. Translating languages requires to map concrete syntactic elements. Manipulating structures deals with abstract modeling concepts like objects or predicates. An important motivation is to separate these different concerns. In particular, the equivalence of constraint formulations is independent from the languages. This argues in favour of a model technical space (MDE TS) gathering modeling concepts and transformation rules and a grammar technical space (Grammar TS) addressing the language issues, as shown in Figure 1.

In the grammar space, models are written in languages given by grammars. In the model space, they are defined as relations between elements that conform to metamodels. The elements are instances of concepts described in metamodels, for example a constraint $x + y = z$ deriving from some algebraic constraint concept. The relations define links between concepts such as composition and inheritance. That allows one to define complex elements, such as constraint systems composed of collections of constraints.

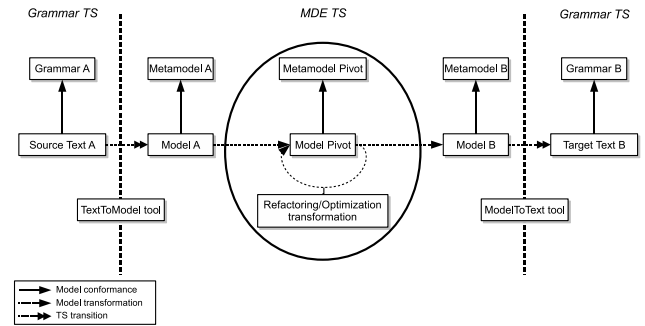


Figure 1: Constraint model transformation process.

The shift from languages to models can be implemented by parsing techniques. Model A is created from the source user model A. This model must conform to the user language metamodel, as is required in the model space. As a consequence, metamodels of languages — modeling languages, constraint programming languages, solver languages — must be defined. The output B is generated from model B. This model must conform to the metamodel of the solver language.

Model transformations are defined in the model space. The goal is to transform model A reflecting the user constraint model to model B associated to the solver. As previously mentioned, that requires to change model representations and structures. This process can be done by rewriting operations manipulating concepts from A to B. In order to share common concepts, we propose to introduce the so-called pivot metamodel. The transformation chain is then a three-steps procedure: a shift from model A to the pivot model, the application of rewriting operations over the pivot model, and a shift from the pivot model to model B.

In the following (Section 4), we will present the pivot metamodels and model transformation operations. However, we will present first a motivating example (next section) and discuss the requirements for handling constraint models.

A Motivating Example

Let us illustrate the transformation process on the social golfers problem. The user model is written in the object-oriented modeling language s-COMMA. The output is a computer program written in the constraint logic programming language ECLⁱPS^e. This problem considers a group of $n = g \times s$ golfers that wish to play golf each week, arranged into g groups of s golfers. The problem is to find a playing schedule for w weeks such that no two golfers play together more than once. Figure 2 and 3 show the s-COMMA model and the ECLⁱPS^e model for this problem, respectively.

The s-COMMA model is divided in a data file and a model file. The data file is composed by an enumeration holding

the golfer names, and three constants to define the problem dimensions (size of groups, number of weeks, and groups per week). The model file is divided into three classes. One to model the groups, one to model the weeks and one to arrange the schedule of the social golfers. The `Group` class owns the `players` attribute corresponding to a set of golfers playing together, each golfer being identified by a name given in the enumeration from the data file. In this class, the code block called `groupSize` (lines 14 to 16) is a constraint zone (constraint zones are used to group statements such as loops, conditionals and constraints under a given name). The `groupSize` constraint zone restricts the size of the golfers group. The `Week` class has an array of `Group` objects and the constraint zone `playOncePerWeek` ensures that each golfer takes part of a unique group per week. Finally, the `SocialGolfers` class has an array of `Week` objects and the constraint zone `differentGroups` states that each golfer never plays two times with the same golfer throughout the considered weeks.

```
//Data file
1. enum Name := {a,b,c,d,e,f,g,h,i};
2. int s := 3; //size of groups
3. int w := 4; //number of weeks
4. int g := 3; //groups per week

//Model file
1. main class SocialGolfers {
2.   Week weekSched[w];
3.   constraint differentGroups {
4.     forall(w1 in 1..w)
5.       forall(w2 in w1+1..w)
6.         forall(g1 in 1..g)
7.           forall(g2 in 1..g) {
8.             card(weekSched[w1].groupSched[g1].players intersect
              weekSched[w2].groupSched[g2].players) <= 1;
9.           }
10.  }
11. }
12. class Group {
13.   Name set players;
14.   constraint groupSize {
15.     card(players) = s;
16.   }
17. }
18. }
19. class Week {
20.   Group groupSched[g];
21.   constraint playOncePerWeek {
22.     forall(g1 in 1..g)
23.       forall(g2 in g1+1..g) {
24.         card(groupSched[g1].players
              intersect groupSched[g2].players) = 0;
25.       }
26.   }
27. }
```

Figure 2: An s-COMMA model of the social golfers problem.

The generated ECLⁱPS^e model is depicted in Figure 3, which has been built as a single predicate whose body is a sequence of atoms. The sequence is made of the problem dimensions (lines 2 to 4), the list of integer sets `L` (lines 6 to 7), and three nested loop blocks resulting from the transformation of the three s-COMMA classes (lines 9 to 36). It turns out that parts of both models are similar. This is due to the sharing of concepts in the underlying metamodels, for instance constants, `forall` statements, or constraints. However, the syntaxes are different and specific processing may be required. For instance, the `for` statement of ECLⁱPS^e needs the `param` keyword to declare parameters defined outside the current scope, e.g. the number of groups `G`.

```
1. socialGolfers(L):-
2.   S $= 3,
3.   W $= 4,
4.   G $= 3,
5.
6.   intsets(WEEKSCHED_GROUPSCHED_PLAYERS,12,1,9),
7.   L = WEEKSCHED_GROUPSCHED_PLAYERS,
8.
9.   (for(W1,1,W),param(L,W,G) do
10.    (for(W2,W1+1,W),param(L,G,W1) do
11.     (for(G1,1,G),param(L,G,W1,W2) do
12.      (for(G2,1,G),param(L,G,W1,W2,G1) do
13.       V1 is G*(W1-1)+G1,nth(V2,V1,L),
14.       V3 is G*(W2-1)+G2,nth(V4,V3,L),
15.       #(V2 /\ V4, V5),V5 $=< 1
16.      )
17.     )
18.    )
19.   ),
20.
21.   (for(I1,1,W),param(L,S,W,G) do
22.    (for(I2,1,G),param(L,S,W,G,I1) do
23.     V6 is G*(I1-1)+I2,nth(V7,V6,L),
24.     #(V7, V8), V8 $= S
25.    )
26.   ),
27.
28.   (for(I1,1,W),param(L,G) do
29.    (for(G1,1,G),param(L,G,I1) do
30.     (for(G2,G1+1,G),param(L,G,I1,G1) do
31.      V9 is G*(I1-1)+G1,nth(V10,V9,L),
32.      V11 is G*(I1-1)+G2,nth(V12,V11,L),
33.      #(V10 /\ V12, 0)
34.     )
35.    )
36.   ),
37.
38.   label_sets(L).
```

Figure 3: The social golfers problem expressed in ECLⁱPS^e.

The treatment of objects is more subtle since they must not participate to ECLⁱPS^e models. Many mapping strategies may be devised, for instance mapping objects to predicates (Soto and Granvilliers 2007). Another mapping strategy is used here, which consists of removing the object-based problem structure. Flattening the problem requires visiting the many classes through their inheritance and composition relations. A few problems to be handled are described as follows. Important impacts on the attributes may happen. For example, the `weekSched` array of `Week` objects defined at line 2 of the model file in Figure 2 is refactored and transformed to the `WEEKSCHED_GROUPSCHED_PLAYERS` flat list stated at line 6 in Figure 3. It may be possible to insert new loops in order to traverse arrays of objects and to post the whole set of constraints. For instance, the last block of `for` loops in the ECLⁱPS^e model (lines 28 to 36) has been built from the `playOncePerWeek` constraint zone of the s-COMMA model, but there is an additional `for` loop (line 28) since the `Week` instances are contained in the `weekSched` array. Another issue is related to lists that cannot be accessed in the same way as arrays in s-COMMA. Thus, local variables (v_i) and the well-known `nth` Prolog predicate are introduced in the ECLⁱPS^e model.

Pivot Model Handling

Our pivot metamodel has been defined to catch most modeling needs that occur in constraint modeling languages. Then, pivot models are managed with several refining transformations, where each transformation identifies a clear refining process, namely structure modifications (e.g. removal

of object variables) or model optimization.

Pivot Metamodel

Figure 4 depicts an extract of our pivot structure metamodel in a simplified UML Class diagram formalism. *Italic font* is used to denote abstract concepts. The root concept is *Model* which contains all entities. Three abstract concepts inherit from the abstract class *ModelElement*:

- *Classifier* represents all types that can be used to define variables or constants:
 - *DataType* corresponds to common primitive data types used in CP, namely Boolean, Integer and Real.
 - *Enumeration* is used to define symbolic types, i.e. a set of symbolic values defined as *EnumLiteral* (not defined here to keep the figure readable), e.g. `enum Name := {a, b, ...}`, line 1 of data file in Figure 2.
 - *Class* is similar to the object-oriented concept of class, but defined in a CP context (Soto and Granvilliers 2007), i.e. a class definition is composed of variable or constant definitions and also constraints and other statements. Thus, a *Class* has a set of features being instances of *ModelFeature*.
- *ModelFeature* corresponds to the instance concepts defined within a model. It is also divided in three concepts:
 - *Record* relates to non-typed instances being composed of a collection of elements, such as tuples. To cover a broader range of record definitions, we define a composition of *ModelFeature* instances.
 - *TypedElement* is an abstract concept corresponding to typed constraint model elements. Thus, it has a reference to a classifier. The concept of array variable is not distinguished from variable, but array can be represented using a sequence of sizes, corresponding to each dimension of an array (more than two dimensions are allowed). These sizes are expressed as *Expression* instances.
 - * *Variable* has an optional *Domain* definition (not shown here) restricting values belonging to the associated type. Three concepts of *Domain* are taken into account: intervals, sets and domains defined as an expression.
 - * *Constant* concept is for constants having a type and a fixed value.
 - *Statement* is used to represent all the other features that may occur in a *Model* or a *Class*:
 - * *Constraint* is the abstract constraint concept having two sub-concepts. *ExpressionConstraint* stands for constraints built inductively from terms and relations. *GlobalCtr* handles global constraints defined by a name and a list of parameters.
 - * *ForAll* defines a loop mechanism over constraints and other statements. It has an iterating variable which is local to the loop.
 - * *If* obviously defines a conditional statement. It is composed of an *Expression* corresponding to the boolean test and two sets of statements corresponding to the statements to take into account according

to the test evaluation. The second set of statements is optional if no alternative to the true evaluation of the test is defined.

- *ParameterizedElement* defines concepts having a list of parameters and not being a classifier neither an instance of a *ModelFeature*:
 - *Predicate* represents logical predicates in a model as in ECLⁱPS^e. Predicates have parameters and a body composed of a sequence of *ModelFeature*, such as variable definitions or constraint statements.
 - *Function* represents user-defined functions stated in a model. It contains also a body, but it is based on a statement used to compute a result.
- The notion of expression is ubiquitous in CP. The related concepts of our metamodel are detailed in Figure 5. They represent all the entities occurring in first-order formulas made from variables, terms, relations, and connectives. The concept *Expression* is abstract and is used as super class for all kinds of expressions:
- *FunctionCall* is used to refer to an already defined *Function* and contains a list of parameters defined as *Expression*.
 - *VarOccurrence* is used to refer to already defined instances: records, variables or constants. It is only composed of a reference to the corresponding instance declaration and to a list of optional indexes to handle arrays. It is specialized in *ObjectOccurrence* in order to express the navigation path to an object attribute (e.g. `groupSched[g1].players`, line 8 in Figure 2). Variable occurrences are not classified according to their declaration type in one of the three expression types inheriting from *Expression* in order to avoid multiple declaration of the same concept, while requiring type inference mechanism.
 - *BooleanExpression* is used to specify boolean concepts occurring in expressions:
 - *BoolValue* represents the terms `true` and `False`.
 - *PredicateCall* corresponds to the call of a predicate with its list of parameters. Thus, it is composed of a reference to a *Predicate* with a list of parameters defined as *Expression* in order to allow at the same time *VarOccurrence* and evaluable expressions, such as `1`, `x` or `x + 1`.
 - *BoolOperator* is an abstract concept having a name representing the symbol of well-known operators. It is specialized in the two common types of operators:
 - * *BoolUnaryOp* corresponds to the negation operator and has an operand corresponding to an *Expression*, since it can be a boolean expression, but also a variable. In the following, operands of all operators will be defined as a composition of *Expression*.
 - * *BoolBinaryOp* corresponds to the several common binary operators returning a boolean value, such as: `↔`, `→`, `and`, `or`, `=`, `≠`, `≤`, `≥`, `<`, `>`.
 - *SetExpression* defines the main constructs available to deal with sets within expressions:

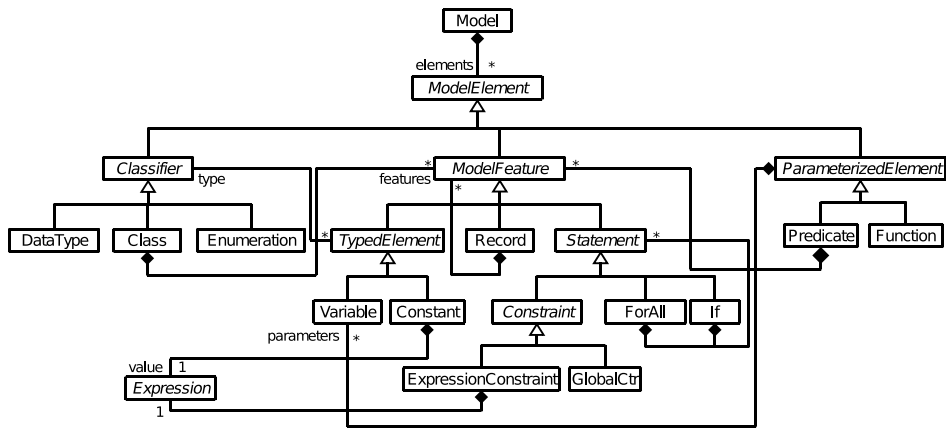


Figure 4: Representation of variables and problem structures in the pivot metamodel.

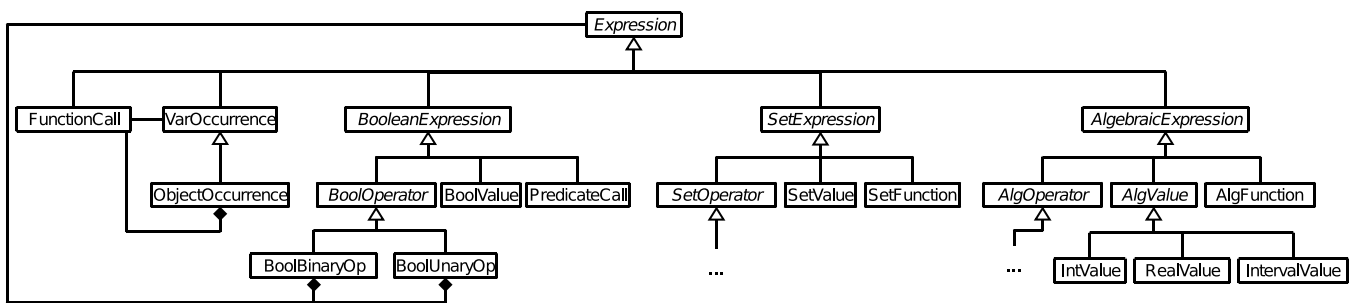


Figure 5: Representation of expressions used to define constraint expression in the pivot metamodel.

- SetValue corresponds to a set of value occurrences, such as $\{1, 2, 3\}$. To tackle various contents as set elements (e.g. $\{1, x + 1\}$), it is composed of a list of elements conformed to Expression.
- SetFunction corresponds to the call of known functions over sets, such as the cardinality function.
- SetOperator is specialized only in SetBinaryOp since no unary operator is commonly used on sets. For instance, intersection, union and difference are available.
- AlgebraicExpression defines the numerical expressions:
 - AlgValue is abstract and represents the three main concepts of values in numerical expressions: IntValue for integer values, RealValue for real number values and IntervalValue for interval values such as $[-1, 1]$.
 - AlgFunction corresponds to the call of a well-known function over numbers, such as trigonometric functions.
 - AlgOperator refers to the common operators used in algebraic expressions: AlgUnaryOp(-, +) and AlgBinaryOp(+, -, *, / and ^).

Our pivot metamodel has been defined to fit with most modeling needs in CP, but also to fit with the metamodel of CP languages. Thus, some simplifications have been done to ease transformations such as the VarOccurrence concept which directly inherits from Expression. Indeed, variable occurrences can be typed in expressions (i.e. boolean, set or algebraic), but we define only one to avoid redundancies.

Pivot model refactoring

Model transformations are implemented as rewriting operations over pivot models.

For sake of clarity, we will present a few operations using an imperative pseudo-code style, while specific transformation languages are used in practice. The main interest given by the concept hierarchy is to provide navigation mechanisms through models. For instance, it is immediate to iterate over the set of variables of a constraint, since this information is gathered in the corresponding abstract constraint concept (see e.g. Algorithm 2). It is therefore possible to manipulate models globally, which is very powerful.

Object flattening This refactoring step replaces object instances, namely variables whose type is a class, by all elements defined in the class definition (variable, constants, constraints and other statements). In order to prevent name conflicts, named elements are prefixed with the name of object instances.

This refactoring transformation can be expressed in terms of a brief pseudo-code algorithm as shown in Algorithm 1. The ObjectRemoval function processes a source model by iterating on all its elements (line 2). If object instances are detected (line 3), then the function flatten is called and its result is added to the output model elements (line 4). Instances not being a Class definition are duplicated

Algorithm 1 Transforming and removing object variables and class definitions

ObjectRemoval($m : \text{Model}$)

: Model

```

1: let res : Model
2: for all o in m.elements do
3:   if is_var(o) and is_class(o.type) then
4:     res.insert(flatten(o,o.type.features))
5:   else if not is_class(o) then
6:     res.elements.insert(o)
7:   end if
8: end for
9: return res

```

flatten($o : \text{Variable}$, features : Set of ModelFeature)

: Set of ModelFeature

```

1: let res : Set of ModelFeature = {}
2: for all f in features do
3:   if is_var(f) and not is_class(f.type) then
4:     let v : Variable
5:     v ← duplicate(f)
6:     v.name = o.name + '_' + v.name
7:     res.insert(v)
8:   else
9:     ...
10:  end if
11: end for
12: return res

```

in the output model (line 5,6), while Class definitions are removed. In the flatten function every feature given as parameter is cloned and added to the resulting set of ModelFeature. In the case of a variable (and also constants), its name is concatenated to the object variable name (line 6). Figure 6 depicts the result of the transformation on the social golfers example previously presented.

```

class SocialGolfers { Week weekSched[w];...}
class Week { Group groupSched[g];...}
class Group { Name set players;...}
⇒ Name set weekSched_groupSched_players[g*w];

```

Figure 6: Applying the object flattening transformation on the social golfers example using s-COMMA syntax.

Arrays of objects and expressions refactoring are not presented here to keep the algorithm simple. As mentioned at the end of Section 3, in the case of object arrays, we must transfer their size to their attributes and a loop statement has to be introduced to iterate on their Statement instances. Within expressions, instances of VarOccurrence may just be updated with the declaration of the new flat variables.

Alldifferent removal Since global constraints are not handled by every solver, there is a motivation to reformulate them or to generate relaxations. We consider here, the well-known global constraint $\text{alldifferent}(x_1, \dots, x_n)$. We assume that the domain of each x_i varies from 1 to n to ease

the definition of the two last algorithms. We propose three possible transformations:

- Generating a set of disequalities as shown in Algorithm 2. For all variable combinations (line 2,3), a constraint is generated and added to the result (line 6).

Algorithm 2 Transforming alldifferent to a set of disequalities

```

AllDiffToDisequalities(c : GlobalConstraint)
: Set of Constraint
1: let res : Set of Constraint =  $\emptyset$ 
2: for all i in 1..c.parameters.size() do
3:   for all j in i + 1..c.parameters.size() do
4:     let x : Variable = c.parameter[i]
5:     let y : Variable = c.parameter[j]
6:     res.insert(new Constraint(x  $\neq$  y))
7:   end for
8: end for
9: return res

```

- Generating a relaxation as shown on Algorithm 3. Only one constraint is created (line 3) assessing that the sum of all variable values is equal to $n(n+1)/2$.

Algorithm 3 Generating alldifferent relaxations

```

AllDiffToRelaxation(c : GlobalConstraint)
: Constraint
1: let n : Integer = c.parameters.size()
2: let sum : Expression =  $\sum_{i=1}^n$  c.parameters[i]
3: return new Constraint(sum = n(n+1)/2)

```

- Generating a boolean version as shown on Algorithm 4. In this case, we define a new matrix of boolean variables (line 2,3,4), where $b[i, j]$ being true means x_i has value j . Line 7 checks that only one value per variable is defined. Line 10 ensures that two variables have different values.

Algorithm 4 Reformulating alldifferent into a boolean model

```

AllDiffToBoolean(c : GlobalConstraint)
: Set of ModelFeature
1: let res : Set of Constraint =  $\emptyset$ 
2: let n : Integer = c.parameters.size()
3: let m : Integer = card(c.parameters.domain)
4: let b[n,m] : Boolean
5: res.insert(b)
6: for i in 1..n do
7:   res.insert(new Constraint( $\sum_{j=1}^m$  b[i,j] = 1))
8: end for
9: for j in 1..m do
10:  res.insert(new Constraint( $\sum_{i=1}^n$  b[i,j] = 1))
11: end for

```

Experiments

The presented architecture has been implemented with three tools and languages: KM3 (Jouault and Bézivin 2006) is a metamodel language, ATL (Kurtev, van den Berg, and Jouault 2007) is a declarative rule language to describe model transformations and TCS (Jouault, Bézivin, and Kurtev 2006) is a declarative language based on templates to define the text to model and model to text transitions. These MDE tools allow us to choose the refactoring steps to apply on pivot models in order to keep supported structures of the target metamodel.

We have carried out a set of tests in order to analyze the performance of our approach. We used five CP problems: Social Golfers, Engine Design, Send+More=Money, Stable Marriage and N-Queens. The first experiment evaluates the performance in terms of translation time, and the second one was done to show that the automatic generation of solver files does not lead to a loss of performance in terms of solving time. The benchmarking study was performed on a 2.66Ghz computer with 2GB RAM running Ubuntu.

Problems	sC Lines	s-to-P (s)	Object (s)	Enum (s)	P-to-E (s)	Total (s)	Ecl Lines
Golfers	31	0.276	0.340	0.080	0.075	0.771	37
Engine	112	0.292	0.641	0.146	0.087	1.166	78
Send	16	0.289	0.273	-	0.089	0.651	21
Marriage	46	0.330	0.469	0.085	0.067	0.951	26
10-Q	14	0.279	0.252	-	0.033	0.564	12

Table 1: Times for complete transformation chains of several classical problems.

In the first experiment we test the s-COMMA (sC) to ECLⁱPS^e (Ecl) translation. Table 1 depicts the results. The first column gives the problem names. The second column shows the number of lines of the s-COMMA source files. The following columns correspond to the time of atomic steps involved in the transformation (in seconds): transformations from s-COMMA to Pivot (s-to-P) (corresponds to Source Text A to Model Pivot in Figure 1), object flattening (Object), enumeration removal (Enum), and transformations from Pivot to ECLⁱPS^e (P-to-E) (corresponds to Model Pivot to Target Text B in Figure 1). The next column details the total time of the complete transformation, and the last column depicts the number of lines of the generated ECLⁱPS^e files.

The results show that the text processing phases (s-to-P and P-to-E) are fast, but we may remark that the given problems are concisely stated (maximum of 112 lines). The transformation s-COMMA to pivot is slower than the transformation pivot to ECLⁱPS^e. This is explained by the refactoring phases performed on the pivot that reduce the number of elements to handle the pivot to ECLⁱPS^e step. The composition flattening is the more expensive phase. In particular, the Engine problem exhibits the slowest running time, since it contains several object compositions. In summary, considering the whole set of phases involved, we believe the results show reasonable translation times.

Problems	Native		Generated		Generated (Flat)	
	solve(s)	Lines	solve(s)	Lines	solve(s)	Lines
Golfers	0.21	28	0.21	31	0.22	276
Marriage	0.01	42	0.01	46	0.01	226
20-Q	4.63	11	4.65	12	5.02	1162
28-Q	80.73	11	80.78	12	87.73	2284

Table 2: Solving times and model sizes of native and generated files

In the second experiment we compare the ECLⁱPS^e files automatically generated by the framework with native ECLⁱPS^e files written by hand (see Table 2). We consider the solving time and the lines of each problem file. The data of the native models is first given. We then introduce generated files where the loops have not been unrolled (avoiding this phase the size of generated solver files is closer to the native ones). In this case, the solving times of both types of files are almost equivalent. At the end, we consider problems including the loop unrolling phase (Flat). This process leads to a considerable increase of model sizes. Only the solving time of the flat 20-Queens and 28-Queens problems are impacted (about 0.4 and 7 seconds). This may be explained by the incremental propagation algorithm commonly implemented in CLP languages. We may suppose that a propagation happens each time a constraint is added to the constraint store. If a for statement is not interleaved with propagation, i.e. it is considered as one block, then only one propagation step is required. This is not the case if loops are unrolled, leading to one propagation for each individual constraint. It results in a slow-down. This negative impact in terms of solving time demonstrates the need for keeping the structure of target models (e.g. not unrolling loops) instead of building a flat model.

Related Work

Model transformation is a recent research topic in CP. Just a few CP model transformation approaches have been proposed. The solver-independent architecture is likely to be the nearest framework to our approach, for instance, MiniZinc (and Zinc), Essence and s-COMMA.

MiniZinc is a high-level constraint modeling language allowing transformations to ECLⁱPS^e and Gecode models. These mappings are implemented by means of Cadmium. The translation process involves an intermediate model where several MiniZinc constructs are replaced by simplified or solver-supported constructs. This facilitates the translation to get a solver model.

Essence is another language involving model transformations. Its solver-independent platform that allows to map Essence models into ECLⁱPS^e and Minion (Gent, Jefferson, and Miguel 2006). A model transformation system called Conjure (Frisch et al. 2005) is included in the framework, which takes as input an Essence specification and refines it to an intermediate language called Essence'. The translation from Essence' to solver code is currently performed by java translators using the tool Tailor.

s-COMMA is an object-oriented language, supported by a

solver-independent platform where solvers can be mapped to ECLⁱPS^e, Gecode/J, RealPaver, and GNU Prolog (Diaz and Codognet 2000). The language also involves an intermediate model called Flat s-COMMA to facilitate the translation. Hand-written translators and MDE-translators have been developed to translate a Flat s-COMMA model in the target solver model.

Our approach can be seen as a natural evolution of this solver-independent architecture. Two major advantages arise. (1) In the aforementioned approaches just one modeling language can be used as the source of the transformation, in our framework many modeling language can be plugged as the source. We believe this enables flexibility and provides freedom to the modelers. (2) In the s-COMMA, MiniZinc, and Essence transformation processes, the refactoring steps (e.g. enumeration removal, loop and set unrolling) are always applied. This makes the structure of the solver file completely different from the original model. In our framework we focus on generating optimized models while trying to maintain as much as possible the original structure of the source model. We believe that keeping the source modeling structures into target models, then improve their readability and understanding.

Conclusion and Future Work

In this paper, we have presented a new framework for constraint model transformations. This framework is supported by an MDE approach and a pivot metamodel that provides independence and flexibility to cope with different languages. The transformation chain involves three main steps: from the source to the pivot model, refining of the pivot model and from the pivot model to the target. Among others, an important feature of this chain is the modularity of mode transformations and that the hard transformation work (refactoring/optimization) is always performed over the pivot. This makes the transformations from/to pivot simpler, and as a consequence the integration of new languages to the architecture requires less effort.

In a near future, we intend to increase the number of CP languages our approach supports. We also want to define more pivot refactoring transformations to optimize and reformulate models. Another major outline for future work is to improve the management of complex CP models transformation chains, which is not investigated in this paper. The order in which refactoring steps are applied and which refactoring step to apply can be automated. However, we may investigate how to qualify models and transformations according to the pivot and target metamodels.

References

- [Apt and Wallace 2007] Apt, K. R., and Wallace, M. 2007. *Constraint Logic Programming using Eclipse*. New York, NY, USA: Cambridge University Press.
- [Chenouard, Granvilliers, and Soto 2008] Chenouard, R.; Granvilliers, L.; and Soto, R. 2008. Model-Driven Constraint Programming. In *ACM SIGPLAN PPDP*, 236–246.
- [Diaz and Codognet 2000] Diaz, D., and Codognet, P.

2000. The GNU Prolog System and its Implementation. In *SAC 2000*, 728–732.

[Duck, Stuckey, and Brand 2006] Duck, G. J.; Stuckey, P. J.; and Brand, S. 2006. ACD Term Rewriting. In *ICLP*, 117–131.

[Frisch et al. 2005] Frisch, A.; Jefferson, C.; Martinez-Hernandez, B.; and Miguel, I. 2005. The Rules of Constraint Modelling. In *IJCAI*, 109–116.

[Frisch et al. 2007] Frisch, A. M.; Grum, M.; Jefferson, C.; Hernández, B. M.; and Miguel, I. 2007. The Design of ESSENCE: A Constraint Language for Specifying Combinatorial Problems. In *IJCAI*, 80–87.

[Frühwirth 2009] Frühwirth, T. 2009. *Constraint Handling Rules*. Cambridge University Press. to appear.

[Gent, Jefferson, and Miguel 2006] Gent, I. P.; Jefferson, C.; and Miguel, I. 2006. Minion: A Fast Scalable Constraint Solver. In *ECAI*, 98–102.

[Granvilliers and Benhamou 2006] Granvilliers, L., and Benhamou, F. 2006. Algorithm 852: RealPaver: an Interval Solver Using Constraint Satisfaction Techniques. *ACM Trans. Math. Softw.* 32(1):138–156.

[Jouault and Bézivin 2006] Jouault, F., and Bézivin, J. 2006. KM3: A DSL for Metamodel Specification. In *FMOODS*, LNCS 4037, 171–185.

[Jouault, Bézivin, and Kurtev 2006] Jouault, F.; Bézivin, J.; and Kurtev, I. 2006. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *ACM GPCE*, 249–254.

[Kurtev, van den Berg, and Jouault 2007] Kurtev, I.; van den Berg, K.; and Jouault, F. 2007. Rule-based Modularization in Model Transformation Languages Illustrated with ATL. *Science of Computer Programming*, 68(3):138–154.

[Muller, Fleurey, and Jézéquel 2005] Muller, P.-A.; Fleurey, F.; and Jézéquel, J.-M. 2005. Weaving Executability into Object-Oriented Meta-Languages. In L. Briand, S. K., ed., *Proceedings of MODELS/UML*, LNCS 3713, 264–278.

[Nethercote et al. 2007] Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards A Standard CP Modelling Language. In *CP*, LNCS 4741, 529–543.

[Puget 1994] Puget, J. 1994. A C++ Implementation of CLP. In *SPICIS*.

[Schulte and Tack 2006] Schulte, C., and Tack, G. 2006. Views and Iterators for Generic Constraint Implementations. In *Recent Advances in Constraints (2005)*, LNCS 3978, 118–132.

[Soto and Granvilliers 2007] Soto, R., and Granvilliers, L. 2007. The Design of COMMA: An Extensible Framework for Mapping Constrained Objects to Native Solver Models. In *IEEE ICTAI*, 243–250.

[Van Hentenryck et al. 1999] Van Hentenryck, P.; Michel, L.; Perron, L.; and Régim, J.-C. 1999. Constraint Programming in OPL. In *PPDP*, LNCS 1702, 98–116.

[Van Hentenryck, Michel, and Deville 1997] Van Hentenryck, P.; Michel, L.; and Deville, Y. 1997. *Numerica: a Modeling Language for Global Optimization*. MIT Press.