# High-Level Reformulation of Constraint Programs

**Brahim Hnich — Pierre Flener**

*Computer Science Division*
*Department of Information Science*
*Uppsala University*
*Box 513*
*S – 751 20 Uppsala*
*Sweden*

{*Brahim.Hnich, Pierre.Flener*}*@dis.uu.se*

ABSTRACT. *We propose a set of reformulation rules for models of constraint satisfaction problems that are written in our high-level constraint programming language* ESRA, *which is more expressive than* OPL *and is compiled into* OPL. *These automatable rules achieve models that are often very similar to what a human modeller would have tried, such as switching from a pure constraint program to an integer linear program. Since, for a given solver and a given instance of a problem, it is very hard to figure out which model is the best, we advocate that tool support of our reformulation rules should operate with a set of training instances. Indeed, this is the only way of guaranteeing that the actually chosen reformulation pays off, at least for instances within the distribution underlying the training instances.*

RÉSUMÉ. *Nous proposons un jeu de règles de reformulation pour modèles de problèmes de satisfaction de contraintes exprimés dans notre langage de haut niveau* ESRA *pour programmation par contraintes, qui est plus expressif qu'*OPL *et qui est compilé en* OPL. *Ces règles automatisables donnent des modèles souvent très similaires à ce qu'un modélisateur humain aurait essayé, comme par exemple le passage d'un pur programme par contraintes à un programme linéaire entier. Comme, pour un solveur donné et une instance donnée d'un problème, il est très difficile de déterminer quel modèle est le meilleur, nous maintenons qu'un outil supportant nos règles de reformulation devrait utiliser un jeu d'instances d'entraînement. En effet, ceci est la seule voie pour garantir que la reformulation effectivement choisie soit rentable, du moins pour les instances dans la distribution sousjacente aux instances d'entraînement.*

KEYWORDS: *Constraint programming, high-level modelling, reformulation.*

MOTS-CLÉS : *Programmation par contraintes, modélisation de haut niveau, reformulation.*

## 1. Introduction

*Constraint satisfaction problems* (CSPs), be they *decision problems* (where appropriate values for the problem variables must be found within their domains, subject to some constraints) or *optimisation problems* (where there also is a numeric cost function that has to be optimised), are very ubiquitous in industry and challenging, as the algorithms needed to solve them efficiently are very complex.

Effective modelling of CSPs is hard, is too much time-consuming, and requires a lot of expertise. It is hard because (*i*) in general CSPs are NP-complete [Mac:77], and (*ii*) the performance of any method that solves them is sensitive to the problem instances [TBK:95, Min:96]. It is too much time-consuming because the modeller is trapped in an iterative process of proposing a model and a solver, transforming them possibly many times if she is not satisfied with the performance, and redoing the whole process all over if the distribution of the instances to be solved is changed. It requires a lot of expertise because the space of possible transformations is huge, and deciding which one to choose is still an art.

The ESRA language introduced in [FHK:01a] is a high-level language for modelling CSPs. It is based on high-level type constructors, such as mappings and permutations. Our ESRA-to-OPL compiler [FH:00] deterministically chooses an appropriate OPL representation for variables of these high-level types, depending on context. Therefore, we can design ESRA-to-ESRA model-reformulation rules that force the compiler to choose one of the other possible OPL representations for these variables. Other reformulation rules add implied constraints, exploit dual viewpoints of high-level types, and so on. These automatable rules achieve models that are often very similar to what a human modeller would have tried, such as switching from a pure constraint program to an integer linear program. We aim at building a practical tool that automates the application of these reformulation rules. The tool should be provided with a set of training instances, so that, given a solver, it can automatically select the best model for the particular distribution underlying these instances.

This paper is organised as follows. In Section 2, we motivate our choice of the ESRA language as the level at which we do reformulations. Then we discuss, in Section 3, our approach to automatic reformulation and compare it to related work on solver/model transformation. In Section 4, some reformulation rules for mappings and permutations, as well as some experimental results, are presented. Finally, in Section 5, we conclude and discuss our directions of future work.

## 2. Overview of the ESRA Language

Starting from the very expressive, declarative, and fast OPL (Optimisation Programming Language) [VH:99], the ESRA language is designed to be even more expressive, and equally fast [FHK:01a]. The ESRA language is in fact a conservative extension of OPL. Like OPL, the ESRA language is strongly typed, and a sugared version of what is essentially a first-order logic language. Unlike OPL, the ESRA language

supports more advanced types such as mappings, and allows variables of these types as well as of type set, making it an actual set constraint language and thus more expressive than OPL. A set of rewrite rules achieves compilation from ESRA into OPL (see [FH:00] for details), yielding models that are often very similar to those that a human OPL modeller would (have to) write anyway, so that there is no loss in solving speed compared to (the available labelling heuristics of) OPL.

In ESRA [FHK:01a], in order to support advanced modelling, powerful high-level type constructors were introduced. The syntax and (informal) meaning of their usage in variable declarations (of the form `var` $\langle Type \rangle$ $\langle Variable \rangle$) is as follows:

– `var {T} S`: Set `S` is a subset of set `T`. A domain of `T` must be known (i.e., either `T` is a domain or `T` is a subset of a domain). The internal representation of sets is hidden from the modeller.

– `var V->W M`: Mapping `M` is from set `V` into set `W`. The domains of `V` and `W` must be known. The internal representation of mappings is hidden from the modeller.

– `var perm(S) A`: Array `A`, indexed by `1..card(S)`, represents a permutation of set `S`. The domain of `S` must be known.

– `var seq(S,K) A`: Array `A`, indexed by `1..K`, represents a sequence, of bounded cardinality `K`, of elements drawn from set `S`. The domain of `S` must be known.

These type constructors make ESRA a more expressive language than OPL.

To illustrate the high expressive power of ESRA, let us have a look at the *Warehouse Location* problem [VH:99], where a company considers opening warehouses on some candidate locations in order to supply its existing stores. Each possible warehouse has the same maintenance cost, and a capacity designating the maximum number of stores that it can supply ($C_1$). Each store must be supplied by exactly one open warehouse ($C_2$). The supply cost to a store depends on the warehouse. The objective is to determine which warehouses to open, and which of them should supply the various stores, such that the sum of the maintenance and supply costs is minimised.

A way of modelling this problem in ESRA is shown in Figure 1. The variable declarations elegantly express that `OpenWarehouses` is a subset of `Warehouses`, and that `Supplier` is a mapping from `Stores` into `OpenWarehouses`. A very natural formulation of the cost function and constraint $C_1$ arises from this, as well as a complete capture of $C_2$ by the variable modelling. (The `...` notation means that actual values are to be read in at run-time from a data file.) The OPL model generated from this ESRA model is in Figure 2 (see [FH:00] for the details of compilation). Note the similarity with the hand-crafted OPL model in [VH:99] (page 178). `OpenWarehouses` now has a 0/1-modelling. Also, `Supplier` now is an array, indexed by `Stores`, with values in `Warehouses` now: constraint $C_1$ therefore now takes a less natural expression, and $C_2$ now appears as an explicit constraint. The ESRA model in Figure 1 is thus higher-level than both that hand-crafted OPL model and our generated OPL model in Figure 2, as it hides representation details from the modeller and allows her to state constraints in a very natural way by the use of very powerful type constructors.

```
int FixedCost = ...;
enum Warehouses ...;
enum Stores ...;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
var {Warehouses} OpenWarehouses;
var Stores->OpenWarehouses Supplier;
minimize
  sum(I->J in Supplier) SupplyCost[I,J]
   + card(OpenWarehouses) * FixedCost
subject to {
  forall(J in OpenWarehouses)
    count(I in Stores: I->J in Supplier) <= Capacity[J] };
```

**Figure 1.** *An* ESRA *model of the Warehouse Location problem*

```
int FixedCost = ...;
enum Warehouses ...;
enum Stores ...;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
var int OpenWarehouses[Warehouses] in 0..1;
var Warehouses Supplier[Stores];
minimize
  sum(I in Stores) SupplyCost[I,Supplier[I]]
   + (sum(J in Warehouses) OpenWarehouses[J]) * FixedCost
subject to {
  forall(I in Stores)
    OpenWarehouses[Supplier[I]]=1;
  forall(J in Warehouses)
    OpenWarehouses[J]=1 =>
      (sum(I in Stores) (Supplier[I]=J)) <= Capacity[J] };
display(I in Warehouses: OpenWarehouses[I]=1) <I>;
```

**Figure 2.** *Generated* OPL *model from the* ESRA *model in Figure 1*

Now, it can be argued that high-level reformulations are very desirable, namely for the following three reasons:

– High-level reformulation rules have rather simple left-hand sides, thus simplifying the matching problem while determining which rules are applicable.

– High-level reformulation rules are less numerous, hence the resulting model space is more manageable.

– High-level reformulation rules are independent of low-level data representation decisions, as the compiler takes care of that.

We thus strongly believe that ESRA models are more suitable for reformulation than their corresponding OPL models, say.

## 3. Program Transformation in Constraint Programming

*Program transformation* is the equivalence-preserving modification of a program into another program, of the *same* programming language, with focus on achieving greater efficiency, in time or space or both. In imperative, object-oriented, functional, and logic programming, many years of widespread intensive efforts have led to a deeper understanding of the complexity of programs and their interaction with the execution mechanisms, to the identification of many useful transformation operators (such as finite differencing, dynamic programming, loop fusion, and so on), and to the design of many practical tools that encourage transformational programming as a software life-cycle (see [Fea:87] for a good survey). These transformation operators, if applicable, come with an optimisation guarantee. However, in the more recent paradigm of constraint programming, not so many results are available, and we survey (below) the ones known to us.

The adaptation of program transformation ideas to constraint programming is not so easy. Indeed, the execution of a constraint program has two phases, with different trade-offs. First, execution of the model of a constraint program just *posts* the constraints to the constraint store, but this is often done in polynomial time and is thus negligible compared to the actual solving time of the problem, if the latter is NP-hard (which is usually the case). Transforming the model — in which case one usually talks of *reformulation* — is a little-understood art, which only recently started gaining the attention it deserves. Its difficulty follows from the interaction of the model with the solver (as shown next). But already note that there is in general almost nothing to be gained from posting the *same* constraints faster! Second, execution of the labelling heuristic of a constraint program (if any, otherwise of the default heuristic of the solver) actually *solves* the problem, using the search and propagation algorithms of the solver. For non-approximate solving, this usually takes non-deterministic polynomial time (NP) and is thus the real bottleneck. Choosing a heuristic and the rest of the solver (namely the search and propagation algorithms) — in which case we here talk of *optimisation* — is a very well-studied field, but also a very frustrating one: the nature of heuristics after all is that they are not guaranteed to perform well in all situations. Experiments [Min:96, TBK:95] confirm this by showing that there not only is no best solver for all problems, but that there even is no best solver for all instances of a given problem, nor for all its models! This means that no guaranteed optimisation can be achieved before solving-time (that is, when the actual instance of the problem is still unknown), and that alternate models need to be considered.

Table 1 compares various existing CSP solver/model transformation approaches, namely Smith's KIDS [Smi:90], Minton's MULTI-TAC [Min:96], Ellman's DA-MSA [EKBA:98], and the reformulation framework proposed by Borrett and Tsang [BT:xx]. The comparison is in terms of five features:

| | KIDS | MULTI-TAC | DA-MSA | B / T | us |
|---|---|---|---|---|---|
| Time | compile | compile | compile | solve | compile |
| Instances | not explicit | distribution | distribution | current one | distribution |
| Object | $M, S, P, H$ | $M, S, P, H$ | $M, S, P, H$ | $M$ | $M$ $(\ , H)$ |
| Choice | manual | automatic | manual | automatic | automatic |
| Control | intuitive | hillclimbing | intuitive | N/A | TBA |

**Table 1.** *Comparison of CSP solver/model transformation tools*

– Transformation can occur at different times, the possibilities being compile-time and solving-time.

– Due to the mentioned sensitivity of solvers to problem instances [TBK:95, Min:96], some transformation tools request a set of training instances reflecting the desired distribution (or a generator producing instances of that distribution), so that a solver/model fine-tuned for instances of that distribution can be sought. Other tools do not ask for an explicit set of training instances, either because navigation through the transformation space is manually steered, by intuition or by experimentation outside the tool, or because transformation occurs at solving-time and is thus necessarily about the current instance.

– The transformed object is either the model only, or the solver only, or both. Constraint solvers are composed of a search algorithm (denoted $S$ here) such as forward-checking, plus a constraint propagation algorithm (denoted $P$ here) based on consistency techniques such as bounds consistency, and labelling heuristics (denoted $H$ here), one of which is the default; the model is denoted $M$ here. When both the model and the (entire) solver are being transformed, then we have transformation of a problem-specific solver. When only the model is being transformed, then the entire solver is fixed (for instance, standard backtracking in the Borrett/Tsang experiments). When the model and part of the solver are being transformed, then the rest of the solver is fixed.

– The choice of which transformation operator (or: *transform*) is applied to which part of the chosen object is either manual (performed by the user, in an interactive tool) or automatic (performed by the tool itself). In both cases, the actual application of the transform is done by the tool, as such error-prone, clerical work should always be left to the machine. In the Borrett/Tsang context, there is (currently) only one reformulation operator, so there is no real choice issue (yet), but their intention seems to be automated choice.

– The search control within the transformation space, and thus the choice of which object to transform next, is either intuitive (leading to a manual choice of the transform) or somehow systematic. For instance, MULTI-TAC performs a hill-climbing beam search; in the Borrett/Tsang framework, there is (currently) only one reformulation operator, so there is no real control issue (yet), as the operator is exhaustively applied to the current instance.

Table 1 also shows that many more (novel) combinations of these features exist. We

advocate the following approach (summarised in the last column of the table), and are following it with the work presented here:

– Considering that solving-time transformation would have to be sufficiently fast for its benefits to *always* offset its overhead, and that achieving this is far from trivial, we opted for compile-time transformation, and even (only) for situations where compilation time does not matter. The underlying hypothesis is thus that the transformed solver/model is run on sufficiently many instances for the compilation time to be offset by gains in solving time. The responsibility for choosing whether (and for how long) to use our transformation tool will thus rest with the user.

– Considering the sensitivity of solvers to problem instances, we opted for training instances of some target distribution to be an explicit input to our transformation tool, so that it can automatically do its own experiments towards evaluating candidate solver/model pairs. These experiments, plus the logging and comparing of their results, are very tedious, but they have to be made, as human intuition breaks down.

– Considering that we want to focus on reformulating models in our ESRA constraint programming language [FHK:01a], we do not aim at solver optimisation and thus fix the search and propagation algorithms to the ones of ESRA. Although ESRA models cannot contain labelling heuristics, we can argue that we also transform these heuristics, as ESRA models are compiled into OPL [VH:99] models and labelling heuristics. The underlying hypothesis is thus that one need not always optimise entire solvers, as their most critical component is an instance-specific labelling heuristic (and new-generation solvers provide primitives for expressing such heuristics).

– Considering the large size of the reformulation space, we prefer automatic choice of which reformulation is to be applied to which part of the chosen model.

– For the same reason, we chose systematic exploration of the reformulation search space, but we have not decided yet which search control to adopt (mainly because we have not identified yet a sufficiently large set of reformulation operators).

Usage of the considered transformation tools, including the one we are building, follows the same generic scenario, shown in Figure 3, where dashed lines indicate optional features. Given (an informal version of) a CSP, a human user formulates a (formal) model thereof in some constraint language. For some tools, the user also prepares a set of training instances of a certain distribution, to which actual instances to be solved later are supposed to belong; this may also be done by providing a generator producing instances of that distribution. Another input that may only exist for some tools consists of *transformation search control parameters*.[1] For instance, a time-interval could be given, after which the best solver/model found that far is to be returned, or a limit on how many transforms should at most be applied, or a limit on the number of models to be considered. These inputs are then passed to the tool, which consists of two modules. First, a *synthesiser* takes the CSP model and returns a solver; this may be an elaborate synthesis of a problem-specific solver, as with KIDS, MULTI-TAC, DA-MSA, or it may simply amount to grabbing an existing solver (pos-

1. It is important not to confuse transformation search and solving search.

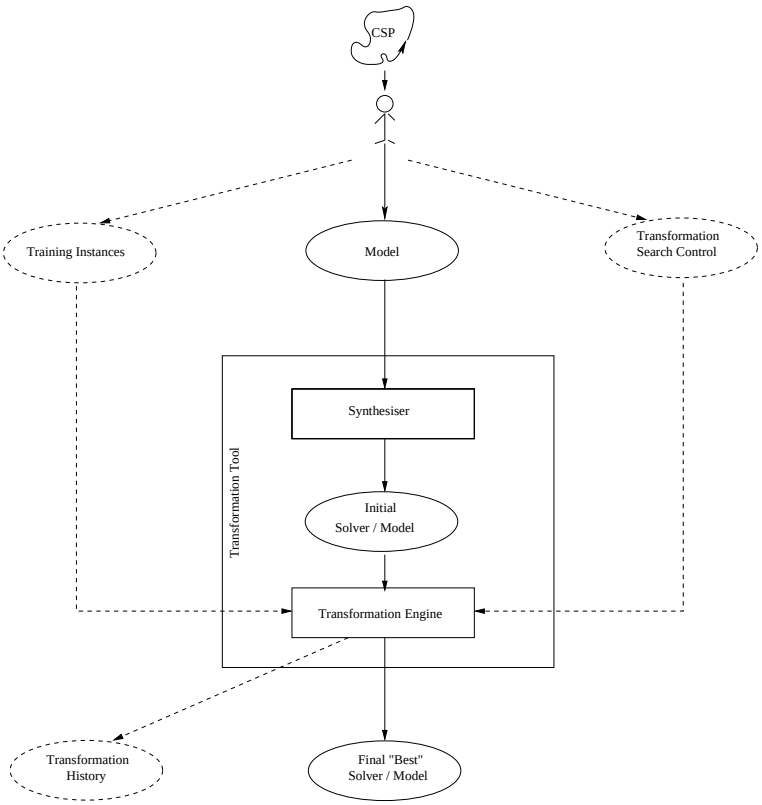


**Figure 3.** *Generic scenario of CSP solver/model transformation*

sibly from a singleton list, as in the Borrett/Tsang experiments). Second, the actual *transformation engine* is given as inputs the obtained solver/model pair, as well as possibly the training instances and the transformation search control parameters. The transformation engine itself consists of at least two parts, namely a *transformer* that chooses and applies a transform on some object, and an *evaluator* that tries an object on the training instances to measure the progress (if any) achieved by a transformation. The transformation engine eventually returns the best solver/model pair it could find within the circumscribed transformation search space, for the training instances. The assumption is that this solver/model pair will perform very well on other instances of the same underlying distribution. Some tools also return a transformation *history*, consisting of the tree of transforms actually applied, and optionally of an isomorphic tree with the solving-cost statistics for the training instances.

The transformation tool we advocate has *all* the features mentioned in Figure 3. Considering our choices in the last column of Table 1, we envisage a synthesiser that simply takes the ESRA solver.

| Case | ESRA declarations | OPL declarations |
|---|---|---|
| 1 | `var V->W M;`<br>where `V` and `W` are domains | `var W M[V];` |
| 2 | `var V->W M;`<br>where `V` is a set variable of domain `S`<br>and `W` is a domain | `var int V[S] in 0..1;`<br>`var W M[S];` |
| 3 | `var V->W M;`<br>where `V` is a domain<br>and `W` is a set variable of domain `T` | `var int W[T] in 0..1;`<br>`var T M[V];` |
| 4 | `var V->W M;`<br>where `V` and `W` are set variables<br>of domains `S` and `T`, respectively | `var int V[S] in 0..1;`<br>`var int W[T] in 0..1;`<br>`var int M[S,T] in 0..1;` |

**Table 2.** OPL *representations of* ESRA *mappings*

## 4. Reformulation Rules for Mappings and Permutations

We now introduce reformulation rules for ESRA mappings and permutations, and explain their motivation. Reformulation rules are here written as $L \Rightarrow R$, meaning that expression $L$ is rewritten into $R$, under a condition identified in the context. For convenience, new variables introduced in $R$ contain an underscore ("_") character.

### 4.1. *Mappings*

At the ESRA level, the mapping type constructor takes two arguments, each of which can be either a domain or a set variable, giving rise to four cases. For each case, the ESRA compiler chooses a different OPL representation, as shown in Table 2, and posts any necessary constraints (omitted here, but see [FH:00] for full details).

For mappings, the reformulation rules (identified so far) force the compiler to choose a different OPL representation. For instance, if we have a mapping `M` from a set variable `V` into a domain `W`, then one can introduce a new set variable `W_s` that is a subset of `W`, and change the mapping to be from `V` into `W_s` instead. This will make the ESRA compiler choose a different OPL representation. However, we need to preserve the semantics of the mapping, and hence add the constraint that `W` is equal to `W_s`. Now we can present all our reformulation rules for mappings by considering each case separately. We omit case 4 as we do not have any rules for it.

**Reformulation Rules for Case 1.** When a mapping `M` is from a domain `V` into a domain `W`, we have the following rules:

```
var V->W M;
  ⇒    var {V} V_s; var V_s->W M; V = V_s;          (M_1^1)
  ⇒    var {W} W_s; var V->W_s M; W = W_s;          (M_1^2)
  ⇒    var {V} V_s; var {W} W_s; var V_s->W_s M;
       V = V_s; W = W_s;                            (M_1^3)
```

| Case | ESRA declarations | OPL declarations |
|---|---|---|
| 1 | `var perm(V) P;` where `V` is a domain | `var V P[1..card(V)];` |
| 2 | `var perm(V) P;` where `V` is a set variable of domain `S` | `var S P[1..card(S)];` `var int P_s[1..card(S)] in 0..1;` |

**Table 3.** OPL *representations of* ESRA *permutations*

The rule $M_1^1$ reformulates `M` as a mapping from the new set variable `V_s` into the domain `W`, where `V_s` is a subset of `V`, but `V = V_s`. Conversely, the rule $M_1^2$ reformulates `M` as a mapping from the domain `V` into the new set variable `W_s`, where `W_s` is a subset of `W`, but `W = W_s`. Combining both options, the rule $M_1^3$ reformulates `M` as a mapping from the new set variable `V_s` into the new set variable `W_s`, where `V_s` (resp. `W_s`) is a subset of `V` (resp. `W`), but `V = V_s` and `W = W_s`.

**Reformulation Rule for Case 2.** When a mapping `M` is from a set variable `V` into a domain `W`, we have the following rule:

```
var V->W M;
  ⇒    var {W} W_s; var V->W_s M; W = W_s;   (M_2^1)
```

The rule $M_2^1$ reformulates `M` as a mapping from the set variable `V` into the new set variable `W_s`, where `W_s` is a subset of `W`, but `W = W_s`.

**Reformulation Rule for Case 3.** When a mapping `M` is from a domain `V` into a set variable `W`, we have the following rule:

```
var V->W M;
  ⇒    var {V} V_s; var V_s->W M; V = V_s;   (M_3^1)
```

The rule $M_3^1$ reformulates `M` as a mapping from the new set variable `V_s` into the set variable `W`, where `V_s` is a subset of `V`, but `V = V_s`.

### 4.2. *Permutations*

At the ESRA level, the permutation type constructor takes one argument, which can be either a domain or a set variable, giving rise to two cases. For each case, the ESRA compiler chooses a different OPL representation, as shown in Table 3, and posts any necessary constraints (omitted here, but see [FH:00] for full details). Furthermore, as discussed in [Gee:92], a permutation over a set `V` has a dual viewpoint, i.e., it can be viewed either as a mapping from `V` into the range `1..card(V)`, or as a mapping from the range `1..card(V)` into `V`, plus a suitable constraint to actually force the mapping to be a bijection, i.e., any two distinct elements of the domain must be mapped to two distinct elements of the co-domain.

For permutations, the reformulations (identified so far) either force the compiler to choose a different OPL representation, or exploit the dual viewpoint of permutations. We here discuss the reformulation rule for case 1, as well as the reformulation rules for the dual viewpoint of permutations.

**Reformulation Rule for Case 1.** The rule:

```
var perm(V) P;
  ⇒ var {V} V_s; var perm(V_s) P; V = V_s;        (P_1^1)
```

reformulates a permutation over a domain `V` as a permutation over the new set variable `V_s`, where `V_s` is a subset of `V`, but `V = V_s`.

**Reformulation Rules Exploiting the Dual Viewpoint.** The following rule captures the reformulation of a permutation over a domain `V` as a mapping from `V` into the range `1..card(V)`:

```
var perm(V) P;
  ⇒    range B_s 1..card(V); var V->B_s P;
       forall(I->J,K->L in P) I<K => J<>L;
AggrOp(i in 1..card(V): Q[i]) F[P[i]]
  ⇒    AggrOp(j_s->i in P) Q[i]*F[j_s]
forall(i in 1..card(V): Q[i]) F[P[i]]                (P_3^1)
  ⇒    forall(j_s->i in P) Q[i] => F[j_s]
exists(i in 1..card(V): Q[i]) F[P[i]]
  ⇒    exists(j_s->i in P) Q[i] & F[j_s]
Constraint[P[i]];
  ⇒    j_s->i in P;
       Constraint[j_s];
```

The previous rules for mappings and permutations need not reformulate any of the constraints as they do not change any type constructors. However, rule $P_3^1$ reformulates the permutation as a mapping and hence all the constraints on the permutation need to be reformulated in terms of the mapping. Therefore, rule $P_3^1$ is composed of different parts that are either all applied or not at all. The first part reformulates the permutation `P` as a mapping from domain `V` into the range `B_s` (which is `1..card(V)`), and forces the mapping to be a bijection. In the second part, $AggrOp$ can be `sum`, `prod`, `min`, or `max`. A summation (resp. product, minimum, maximum) over the elements of a permutation is reformulated as a summation (resp. product, minimum, maximum) over the elements of the mapping. The third and fourth parts reformulate a universal (resp. existential) quantification over the elements of a permutation as a universal (resp. existential) quantification over the elements of the mapping. Note that $F$ can be any ESRA constraint, while $Q$ can be any ESRA relation. The last part reformulates any non-aggregate constraint involving `P[i]` into a constraint on `j_s`, where `i` is the image of `j_s` under the mapping `P`.

The following rule captures the reformulation of a permutation over a domain `V` as a mapping from the range `1..card(V)` into `V`:

```
var perm(V) P;
  ⇒    range B_s 1..card(V); var B_s->V P;
       forall(I->J,K->L in P) I<K => J<>L;
AggrOp(i in 1..card(V): Q[i]) F[P[i]]
  ⇒    AggrOp(i->j_s in P) Q[i]*F[j_s]
forall(i in 1..card(V): Q[i]) F[P[i]]                    (P_3^2)
  ⇒    forall(i->j_s in P) Q[i] => F[j_s]
exists(i in 1..card(V): Q[i]) F[P[i]]
  ⇒    exists(i->j_s in P) Q[i] & F[j_s]
Constraint[P[i]];
  ⇒    i->j_s in P;
       Constraint[j_s];
```

Similarly to rule $P_3^1$, the rule $P_3^2$ is composed of different parts. The first part reformulates the permutation `P` as a mapping from range `B_s` into the domain `V`, and forces the mapping to be a bijection. The second to fourth parts are analoguous to rule $P_3^1$. The last part reformulates any non-aggregate constraint involving `P[i]` into a constraint on `j_s`, where `j_s` is the image of `i` under the mapping `P`.

### 4.3. *Experimental Results*

For the Warehouse Location problem, the ESRA model in Figure 1 uses a mapping from a domain (`Stores`) into a set variable (`Openwarehouses`). Thus, the reformulation rule $M_3^1$ can be applied. In Figure 4 is the resulting ESRA model after applying $M_3^1$. The OPL model generated from the ESRA model in Figure 4 is shown in Figure 5. The constraints of the generated OPL model in Figure 5 can be simplified (manually for the time being), resulting in constraints (shown in Figure 6) very similar to the ones of the hand-crafted OPL model in [VH:99] (page 161, which mistakenly lacks the second constraint). The simplification is as follows. The second and third constraints in Figure 5 are simplified to the second and third constraints in Figure 6 because all the Booleans of the set variable `Stores_s` are set to 1, and `A=1 => B=1` is equivalent to `A <= B`. The last constraint in Figure 5 is simplified to the last constraint in Figure 6 because `Supplier[I,J]=1` implies that `OpenWarehouses[J]=1`.

Note also that the generated OPL model from the ESRA model in Figure 1 is a constraint program (CP), while the simplified version of the generated OPL model from the ESRA model in Figure 4 is a pure 0-1 integer linear program (ILP). We experimented with the instance data provided in [VH:99] (page 162), and the reformulation paid off in terms of solving times, as the new model is 10 times faster.

The reader should not be misled with the results of such experiments, as they just show that the performance of any model is instance-dependent, but not that our reformulated models are always better.

```
int FixedCost = ...;
enum Warehouses ...;
enum Stores ...;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
var {Stores} Stores_s;
var {Warehouses} OpenWarehouses;
var Stores_s->OpenWarehouses Supplier;
minimize
  sum(I->J in Supplier) SupplyCost[I,J]
   + card(OpenWarehouses) * FixedCost
subject to {
  Stores subset Stores_s;
  forall(J in OpenWarehouses)
    count(I in Stores: I->J in Supplier) <= Capacity[J] };
```

**Figure 4.** *A reformulated* ESRA *model of the one in Figure 1*

```
int FixedCost = ...;
enum Warehouses ...;
enum Stores ...;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores,Warehouses] = ...;
var int Stores_s[Stores] in 0..1;
var int OpenWarehouses[Warehouses] in 0..1;
var int Supplier[Stores,Warehouses] in 0..1;
minimize
  sum(I in Stores, J in Warehouses) SupplyCost[I,J] * Supplier[I,J]
   + (sum(J in Warehouses) OpenWarehouses[J]) * FixedCost
subject to {
  forall(I in Stores) Stores_s[I] = 1;
  forall(I in Stores, J in Warehouses)
    Supplier[I,J]=1 => (OpenWarehouses[J] = 1 & Stores_s[I] = 1);
  forall(I in Stores)
    Stores_s[I] = 1 => (sum(J in Warehouses) Supplier[I,J]) = 1;
  forall(J in Warehouses)
    (OpenWarehouses[J]=1) =>
      (sum(I in Stores) (Supplier[I,J]=1)) <= Capacity[J] };
display(I in Warehouses: OpenWarehouses[I]=1) <I>;
display(I in Stores, J in Warehouses: Supplier[I,J]=1) <I,J>;
```

**Figure 5.** *Generated* OPL *model from the* ESRA *model in Figure 4*

The $N$-queens problem can be modelled as a permutation over the set of queens, each of them being assigned to a column of the chessboard by the data modelling. By

```
forall(I in Stores) Stores_s[I] = 1;
forall(I in Stores, J in Warehouses)
  Supplier[I,J] <= OpenWarehouses[J];
forall(I in Stores)
  (sum(J in Warehouses) Supplier[I,J]) = 1;
forall(J in Warehouses)
  (sum(I in Stores) (Supplier[I,J]=1)) <= Capacity[J];
```

**Figure 6.** *Simplified constraints of the* OPL *model in Figure 5*

applying rules $P_3^1$ and $P_3^2$, the resulting models involve a mapping that either maps queens (i.e., columns) to rows, or maps rows to queens (i.e., columns). For each of these reformulated models, the reformulation rules $M_1^1$, $M_1^2$, and $M_1^3$ can then be applied. For instance, after the application of $M_1^3$, we get after simplification a pure 0-1 integer linear program, which views the $N$-queens problem as assigning Booleans to the squares of the chessboard, such that assigning 1 to a Boolean means that we must place a queen in its corresponding square and assigning 0 means the opposite.

## 5. Conclusion

Our main contribution is a set of automatable rules (for mappings and permutations) for reformulating CSP models written in a high-level constraint programming language, such as our ESRA. After compilation into OPL of the resulting ESRA models, and after simplification of these OPL models, we often get what a human OPL modeller would have tried anyway, such as switching from a pure constraint program to an integer linear program. The key idea is the choice of the right level at which we do the reformulations, namely at the ESRA level.

As some of our reformulation rules basically force the ESRA-to-OPL compiler to choose a different internal representation of a high-level data structure, it is clear that the same effect could have been achieved by making the compiler non-deterministic. However, we prefer to have a deterministic compiler, as this is the traditional view of compilers and as such a non-deterministic compiler would be unable to make a recommendation about when each of its output programs is preferable. Helping with this choice is rather the task of a programming environment, taking a set of training instances as further input, and our advocated reformulation tool proposes just that. Indeed, notice that our other rules propose 'classical' reformulations.

Related to our work are Smith's KIDS [Smi:90], Minton's MULTI-TAC [Min:96], Ellman's DA-MSA [EKBA:98], and the reformulation framework proposed by Borrett and Tsang [BT:xx]. A comparison between these approaches and ours has been presented in Section 3.

As future work, we will implement a prototype reformulation tool, working under the scenario shown in Section 3. We also need to enhance our ESRA compilation ma-

chinery, so that the generated OPL models are automatically simplified, thus enabling the full effect of the reformulations. We will furthermore continue searching for reformulation rules, for mappings and permutations, as well as for sequences and subsets. Finally, since the model in Figure 2 is a pure CP model and the model in Figure 6 is a pure ILP model, we will investigate the possibility of reformulation rules that integrate CP and ILP models at the ESRA level so as to generate hybrid ESRA models.

Acknowledgements

This research is partly funded under grant number 221-99-369 of TFR (the Swedish Research Council for Engineering Sciences). We would also like to acknowledge our colleague Zeynep Kızıltan for fruitful discussions and useful comments, as well as the anonymous referees for their useful comments.

## 6. References

[BT:xx] J.E. Borrett and E.P.K. Tsang. A context for constraint satisfaction problem formulation selection. *Constraints*, forthcoming.

[EKBA:98] T. Ellman, J. Keane, A. Banerjee, and G. Armhold. A transformation system for interactive reformulation of design optimization strategies. *Research in Engineering Design* 10(1):30–61, 1998.

[Fea:87] M.S. Feather. A survey and classification of some program transformation approaches and techniques. In: L.G.L.T. Meertens (ed), *Program Specification and Transformation*, pp. 165–195. Elsevier, 1987.

[FH:00] P. Flener and B. Hnich. The syntax and semantics of ESRA. ASTRA Internal Report. Available via http://www.dis.uu.se/∼pierref/astra/.

[FHK:01a] P. Flener, B. Hnich, and Z. Kızıltan. Compiling high-level type constructors in constraint programming. In: I.V. Ramakrishnan (ed), *Proc. of PADL'01*. LNCS. Springer-Verlag, 2001.

[Gee:92] P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proc. of ECAI'92*, pp. 31–35. John Wiley & Sons, 1992.

[Mac:77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence* 8(1):99–118, 1977.

[Min:96] S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints* 1(1–2):7–43, 1996.

[TBK:95] E.P.K. Tsang, J.E. Borrett, & A.C.M. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. *Proc. of AISB'95*, pp. 203–216. IOS Press.

[Smi:90] D.R. Smith. KIDS: A semi-automatic program development system. *IEEE Trans. on Software Engineering* 16(9):1024–1043, 1990.

[VH:99] P. Van Hentenryck. *The* OPL *Optimization Programming Language*. MIT Press, 1999.