Towards Relational Modelling of Combinatorial Optimisation Problems

Pierre Flener
Department of Information Science
Uppsala University, Box 513, S – 751 20 Uppsala, Sweden
Pierre.Flener@dis.uu.se

Abstract

A high-level abstract-datatype-based constraint modelling language opens the door to an automatable empirical determination — by a compiler — of how to 'best' represent the variables of a combinatorial optimisation problem, based on (real-life) training instances of the problem. In the extreme case where no such training instances are provided, such a compiler would simply be non-deterministic. A first-order relational calculus with sets is a good candidate for such a language, as it gives rise to very natural and easy-to-maintain models of combinatorial optimisation problems.

1 Introduction

In recent years, modelling languages based on some logic with sets and relations have gained popularity in formal methods, witness the B [1] and Z [13] specification languages, the ALLOY [9] object modelling language, and the Object Constraint Language (OCL) of UML. In database modelling, this had been long advocated, most notably via entity-relation-attribute (ERA) diagrams. This study examines whether constraint modelling can benefit from the same ideas.

Sets and set expressions recently started appearing as modelling devices in some constraint programming languages, with set variables often implemented by the set interval representation [7]. In the absence of such an explicit set concept, modellers usually represent a set variable as an array of 0/1 integer variables, indexed by the domain of the set. In terms of propagation, the set interval representation is equivalent to the 0/1 representation. The latter of course consumes more memory, but is able to support more set expressions and constraints. Both options are restricted to finite sets.

Relations have not received much attention yet in constraint programming languages, except the particular case of a total function, via the concept of array. Indeed, a total function f can be represented either as a 1D array of variables over the range of f, indexed by its domain, or as a 2D array of 0/1 variables, indexed by the domain *and* range of f, or even with some redundancy. Alternatively, we have recently advocated that total functions should be supported by an abstract datatype, so that it is the compiler that must choose a suitable representation, depending on the context [4] and given (real-life) training instances [8]. Other than retrieving the (unique) image under a total function of a domain element, there has (thus) been no support of relational expressions.

I here make two claims. First, a high-level abstract-datatype-based constraint modelling language opens the door to an automatable empirical determination — by a compiler — of how to 'best' represent the variables of a combinatorial optimisation problem, based on (real-life) training instances of the problem. In the extreme case where no such training instances are provided, such a compiler would simply be non-deterministic. Second, our previous work on an abstract datatype for total functions [4; 8] can be usefully generalised to support *any* kind of relations, and a suitable first-order relational calculus with sets is a good candidate for such a language, as it gives rise to very natural and easy-to-maintain models of combinatorial optimisation problems.

I here ignore the issue of how to parameterise a solver, say by providing a suitable labelling heuristic, towards the solving of the modelled problem. For non-expert or lazy modellers, this task can also be left to compilers [5; 11]. I thus here only aim at techniques that find the 'best' model for a *given* solver, under its *default* settings. A joint consideration of the modelling and the solver parameterisation is omitted here so as not to clutter the reported ideas.

As befits a workshop, this is partially a position paper, in the sense that the denotational and operational formal semantics of the proposed language have not been fully worked out yet (but see [3]), and that a prototype of the advocated compiler is not available yet. My aims here only are to present the notation, illustrate its elegance and the flexibility of its models by some examples, and argue that the advocated compilation philosophy is feasible and useful.

This paper is organised as follows. In Section 2, I present my relational notation for modelling combinatorial optimisation problems. Next, I illustrate my claims on two such problems, namely the Warehouse Location problem in Section 3 and the Stable Marriage problem in Section 4. Finally, in Section 5, I discuss related and future work.

2 Relational Modelling with ESRA

After discussing, in Section 2.1, the design decisions behind the new ESRA modelling language, I introduce, in Section 2.2, its concepts, syntax, and semantics.

2.1 Design Decisions

In constraint satisfaction, much more effort has been directed at efficiently solving the constraints than at facilitating their modelling. Constraint programming languages reflect this, as their control structures and variable representation options are usually quite low-level. This has significantly changed with the advent of the Optimisation Programming Language (OPL) [14], which provides a (nicely sugared) bigger subset of first-order logic than the usual Horn clauses, plus enumerated sets for dispensing with the frequent need of encoding everything as integers, and direct-access arrays (of any dimension) instead of the usual sequential-access lists. But OPL does not go far enough as a modelling language, in our opinion: set variables, relations, and some useful quantifiers are still missing, and thus need to be represented at a rather low level, in unnatural *but standard* ways (see below for examples). The solver of OPL is excellent, especially due to its being a front-end to *both* constraint and linear programming solvers, though a more open parameterisation would be useful.

The key design decisions for our new constraint modelling language — called ESRA — are as follows. We want to capture common modelling idioms in new abstract datatypes and quantifiers, especially for sets and relations, so as to design a truly high-level language. Computational completeness is not aimed at, as long as the notation is useful for elegantly modelling a large number of combinatorial optimisation problems. Like OPL, we (currently) do not support procedures, and hence no procedure calls and no recursion. Similarly, like OPL, we focus on finite domains, and support only bounded quantification. In order to maximally sugar the first-order-logic nature of the language, we adopt an OPL/ALLOY-like 'lower-128 ASCII' syntax, unlike the LaTeX-requiring syntax of Z, as well as an OPL-like JAVA-style declaration of the universally quantified input/output parameters.

Considering the excellent starting point that OPL thus provides, it became natural to design ESRA by extending a streamlined (significant) subset of OPL.[1] The semantics[2] of ESRA will be given in an implementation-independent way, in two layers. Indeed, some features of ESRA are just syntactic sugar for combinations of (a few) kernel features, hence we will provide an operational semantics (by rewrite rules) for the non-kernel features, and a set-oriented denotational semantics for the kernel features. As shown in [3; 4], it is actually possible to give an operational semantics by ESRA-to-OPL rewrite rules for the *entire* language.

2.2 Concepts, Syntax, and Semantics of ESRA

For reasons of space, I here only introduce the concepts of ESRA that are actually illustrated in this paper. Also, I can "only" give an informal semantics. The reader may monitor [3] for a complete description of the full language.

Modelling the Instance Data and Variables. A *primitive type* is either a finite enumeration of new constant identifiers, or a finite range of integers, indicated by its lower and upper bounds. Constant and variable identifiers can be any mix of lowercase and uppercase letters. The only predefined primitive types are the ranges `nat` and `int`, which are `0:maxint` and `-maxint:maxint`, respectively, with `maxint` being the maximum representable integer.

An enumeration is viewed as a *set*, and can thus have subsets. For this, the binary $\{\cdot\}$ type constructor allows the construction of the *powerset* of a set as a new type, so that sets can be declared of that type. Consider the powerset type `{S m:n}`. Then `S` must be a primitive type of the enumeration kind, while the range `m:n` is a *multiplicity*, stating that any member of the powerset of `S` must have between `m` and `n` elements, where `m` and `n` are natural-number expressions.

Relations are declared using the quaternary # relation type-constructor. Consider the relation type `A m:n # p:q B`. Then `A` and `B` must be primitive types, designating the two participants of any relation of this type, with `A` being called the *domain* and `B` the *range* of such a relation, by extension of the terminology for functions, which are just particular cases of relations. The second and third arguments of # are multiplicities, with the following semantics: for every element of `A`, there are between `m` and `n` elements of `B`, and for every element of `B`, there are between `p` and `q` elements of `A` in such a relation.[3] For partial and total functions, `m:n` is `0:1` and `1:1`, respectively. For injections, surjections, and bijections, `p:q` is `0:1`, `1:maxint`, and `1:1`, respectively. Rather than elevating functions and their particular cases to first-class concepts with specific syntax in ESRA, I prefer keeping the notation lean and leave their specialised handling to the compiler. This has the further advantage that only the multiplicities need to be changed during model maintenance when a function becomes a relation (as seen in Section 4.3).

Like in OPL, (arrays of) instance data variables are declared in a JAVA-style strongly typed syntax. Unlike in OPL, *all* instance data are here read in at run-time from a data file.[4] Problem variable declarations follow the same syntax, but are preceded by the `var` keyword. The usage of arrays of problem variables, though possible, is sometimes discouraged, as they sometimes amount to a premature commitment to a low-level representation of what essentially are relation variables. Due to the (current) restrictions on relations, arrays are *not* a redundant feature of ESRA. All declarations denote universally quantified variables, with the instance data ones expected to be ground at solving-time and the problem ones expected to still be variables then. Consider the following declarations:

```
nat MaxWives
enum Women, Men
{Men 1:maxint} MarriedMen
```

[1] OPL leaves numerous opportunities for reducing its syntax while increasing its power. For time reasons, we do *not* support floats, scheduling, and some other features of OPL.

[2] OPL does *not* (yet) have a published semantics.

[3] I thus (currently) restrict the focus to *binary* relations, between primitive types only. My convention is the opposite of the UML/ALLOY one, say, where multiplicities are written in the other order, with the *same* semantics. That convention can however *not* be usefully upgraded to relations of arity higher than 2, and I wanted to leave the possibility for that extension open.

[4] I thus dispensed with all other OPL forms of initialisation, and thus with its '`...`' notation for data-file initialisation. I also dispensed with its semicolon '`;`' after declarations.

```
int RankW[Women,Men]
var Women f
var Women 0:1 # 0:MaxWives Men Marriage
```

Here, `MaxWives` is declared to be a natural number, while `Women` and `Men` are enumerated sets. The set `MarriedMen` is an element of the powerset of `Men`, that is a subset of `Men`, and must be non-empty. The 2D integer array `RankW` is indexed by the sets `Women` and `Men`. The problem variable `f` designates an element of the set `Women`. Finally, the problem variable `Marriage` designates a relation over the sets `Women` and `Men`, such that there is at most one husband for each woman, and at most `MaxWives` wives for each man.

Modelling the Cost Function and the Constraints. Expressions are constructed in the usual way. For *numeric expressions*, arguments are either integers, or the constant `maxint`, or variables of type `nat` or `int`, and the usual unary, binary, and aggregate arithmetic operators are available, such as `card` for the cardinality of a set expression, `ord` for the position of an identifier in an enumeration, the infix '`+`' and '`*`' for the addition and multiplication of two numeric expressions, and `sum` for the sum of a bounded (and possibly filtered) number of numeric expressions.

For *set expressions*, arguments are either enumerated sets or set variables, and the usual binary and aggregate set operators are available, such as the infix `union` for the union of two set expressions, and navigation expressions, explained next. Let `R` be a relation of type `A m:n # p:q B`. For any element (or subset) `a` of `A`, the *navigation expression* `a.R` designates the relational image of `a`, that is the possibly empty set of all elements in `B` that are related by `R` to (any element in) `a`. If `m:n` is `1:1`, then `a.R` simply designates the (unique) element of `B` that is related to element `a` of `A`. The *relation expression* `~R` designates the *transpose* relation of `R`, which is thus of type `B p:q # m:n A`. Transitive closure will be added if suitable examples justify it. The elements of a relation are represented as `a#b` pairs.

First-order logic formulas are also constructed in the usual way, with some restrictions. Atoms are built from expressions with the usual predicates, such as the infix `in` for set or relation membership and the infix '`<=`' for the '$\leq$' inequality between numeric expressions. Formulas are built from atoms with the usual connectives and quantifiers, such as `not` for negation, the infix '`&`' and '`=>`' for conjunction and implication, and `forall` and `exists` for bounded (and possibly filtered) universal and existential quantification. The usual typing and precedence rules for operators and predicates apply. All binary operators associate to the left.

The *cost function* is a numeric expression that has to be either minimised or maximised. The *constraints* on the problem variables are a conjunction of formulas. Consider the following three constraints, given the declarations above:

```
card(Women.Marriage)
  < card(Men.~Marriage)
forall(m in Men)
  card(m.~Marriage) < MaxWives
exists(h in Men: f#h in Marriage)
  card(h.~Marriage) = 1
```

In the first constraint, the navigation expression `Women.Marriage` designates the set of married men, while `Men.~Marriage` designates the set of married women, which must thus be larger than the previous set. The second constraint requires that no man be actually married to the maximum legal number of wives. The third constraint says that woman `f` must be married, to a man `h` who is monogamous. Note the finite bounds '`in Men`' and the optional filters (with problem variables), such as '`f#h in Marriage`', in these quantifications.

```
int MaintCost = ...;
enum WareHs ...;
enum Stores ...;
int Capacity[WareHs] = ...;
int SupplyCost[Stores,WareHs] = ...;
var WareHs Supply[Stores];            // C1
var int OpenWareHs[WareHs] in 0..1;
minimize
 sum(s in Stores) SupplyCost[s,Supply[s]]
  + sum(w in WareHs) OpenWareHs[w]
      * MaintCost
subject to {
 forall(s in Stores)                  // link
   OpenWareHs[Supply[s]]=1;
 forall(w in WareHs)                    // C2
   (sum(s in Stores) (Supply[s]=w))
     <= Capacity[w]
};
```

Figure 1: Published OPL model of the Warehouse problem

3 The Warehouse Location Problem

In the *Warehouse Location* problem, a company considers opening warehouses on some candidate locations in order to supply its existing stores. Each candidate warehouse has the same maintenance cost, and the supply cost to a store depends on the warehouse. Each store must be supplied by exactly one open warehouse (C_1). Each candidate warehouse has a capacity designating the maximum number of stores that it can supply (C_2). The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimised. In more mathematical terms, the sought supply relationship is a *total function* (or: *mapping*) from the set of stores into the set of warehouses, and the set of warehouses to be opened is the *range* of that function.

A First OPL Model. This problem was first modelled as a constraint program in the reference manual of ILOG SOLVER 4.0 (in 1997), and then modelled in OPL in [14]: see Figure 1 for (a syntactic and re-typeset variant of) that model. No prior knowledge of OPL is assumed here, as I explain all its features used here. The maintenance cost is an integer read in at run-time from a data file (which is indicated by the '`...`' notation), and so are the enumerated sets of candidate warehouse locations and existing stores, the 1D array with the integer capacities of the warehouses, and the 2D array with the

```
int MaintCost = ...;
enum WareHs ...;
enum Stores ...;
int Capacity[WareHs] = ...;
int SupplyCost[Stores,WareHs] = ...;
var WareHs Supply[Stores];            // C1
minimize
 sum(s in Stores) SupplyCost[s,Supply[s]]
  + sum(w in WareHs)
     (sum(s in Stores) (Supply[s]=w)>0)
       * MaintCost
subject to {
 forall(w in WareHs)                  // C2
   (sum(s in Stores) (Supply[s]=w))
     <= Capacity[w]
};
```

Figure 2: A second OPL model of the Warehouse problem

integer supply costs to the stores from the warehouses. The sought total function is modelled by a 1D array of variables representing the (unique) warehouse that supplies each store, so this choice fully captures constraint C_1. The set of warehouses to be opened is modelled in a redundant way (because it would suffice to retrieve the range of that function), namely as a 1D array of 0/1 variables, such that `OpenWareHs[w]` is `1` if and only if warehouse `w` is opened.[5] The objective statement expresses that the addition of the sum of the actually incurred supply costs and the sum of the maintenance costs for the actually opened warehouses must be minimal. The second sum is awkward, as the Booleans of `OpenWareHs` are re-interpreted as numeric weights. The first constraint is a *linking* (or: *channelling*) *constraint* made necessary by the redundant modelling. It expresses that a warehouse that is actually supplying some store must be opened. The second constraint captures C_2 using the *higher-order constraint* feature of OPL, whereby a constraint wrapped in parentheses, such as `(Supply[s]=w)`, is associated with a 0/1 integer variable that becomes `1` when the constraint can be proven true, and `0` when the constraint can be proven false, but that remains undetermined otherwise. This may be a very powerful feature of OPL, but reflects low-level thinking and should be hidden from the modeller via a more high-level syntax.

A Second OPL Model. Let us design another OPL model, which non-redundantly models the supply function, namely just by the 1D array of variables representing the warehouse that supplies each store. Figure 2 has the resulting model. The linking constraint disappeared and the second sum of the cost function became much more awkward, as `OpenWareHs[w]` had to be replaced by a higher-order expression returning `1` iff there exists a store supplied by warehouse `w`. On the instance data I tried, this model is an order of magnitude more efficient (by all measures) than the one above. This proves that redundancy elimination may pay off, but it may just as well be the converse, as shown in Section 4.1. But this is hard to guess, as human intuition may be weak here.

[5] OPL does *not* allow set variables, unlike ILOG SOLVER.

```
nat MaintCost
enum WareHs, Stores
nat Capacity[WareHs],
    SupplyCost[Stores,WareHs]
var Stores 1:1 # nat WareHs Supply // C1
minimise
 sum(s#w in Supply) SupplyCost[s,w]
  + card(Stores.Supply) * MaintCost
subject to {
 forall(w in WareHs)               // C2
   card(w.~Supply) <= Capacity[w]
}
```

Figure 3: An ESRA model of the Warehouse problem

An ESRA Model. Figure 3 shows an ESRA model of the problem. It is very different from the one we previously gave [4], as it uses navigation expressions and has no redundancy. The modelling of the instance data is similar to the OPL models above. The sought supply relationship is modelled as a relation and constrained to be a total function from the stores into the warehouses, thereby capturing constraint C_1. The elegance of the cost function reflects the freedom from representation choices, with the navigation expression `Stores.Supply` retrieving the set of warehouses that are to be opened. The only constraint gracefully captures C_2, using the navigation expression `w.∼Supply` to retrieve the set of stores that warehouse `w` supplies. This model can be translated into various OPL models, similar to the ones above.

4 The Stable Marriage Problem

I now consider the Stable Marriage problem, first in its original version (in Section 4.1) and then in two modified versions (in Sections 4.2 and 4.3). For the original version, I discuss three OPL models and exhibit a single ESRA model, which could be mechanically compiled into OPL models not unlike the ones given. For the modified versions, I only exhibit ESRA models, showcasing the flexibility of the proposed relational language, and leaving model maintenance at a lower level (such as OPL) as an exercise to the reader.

4.1 Original Version

The original version of the *Stable Marriage* problem can be stated as follows. Consider a dating agency where an equal number n of women and men have signed up and are willing (for whatever reason) to marry any opposite-sex person of the group. They have ranked all possible spouses by decreasing preference. Figure 4 has sample instance data, where a lower rank means a higher preference. For instance, Hal is Nat's first choice, but it is Eve who is Hal's first choice. The objective is to match up the women and men such that all marriages are stable. A marriage is said to be *stable* if, whenever spouse s prefers some other partner, this partner prefers her/his spouse to s. This means that s may be unhappy, but s/he is bound to stay with her/his spouse. In more mathematical terms, the sought marriages form a *bijection* between the sets of women and men.

	Hal	Jim	Bob	Ian	Guy
Nat	1	2	4	3	5
Eve	3	5	1	2	4
Pat	5	4	2	1	3
Sue	1	3	5	4	2
Val	4	2	3	5	1

	Nat	Eve	Pat	Sue	Val
Hal	5	1	2	4	3
Jim	4	1	3	2	5
Bob	5	3	2	4	1
Ian	1	5	4	3	2
Guy	4	3	2	1	5

Figure 4: Rankings of the women for the men (top), and rankings of the men for the women (bottom)

```
enum Women ...;
enum Men ...;
int RankW[Women,Men] = ...;
int RankM[Men,Women] = ...;
var Men Husband[Women];// all women marry
var Women  Wife[Men];  // all   men marry
solve {
 forall(w in Women)  // link 1: bijection
   Wife[Husband[w]] = w;
 forall(m in   Men)  // link 2: bijection
   Husband[Wife[m]] = m;
 // stability 1:
 forall(w in Women, o in Men)
   RankW[w,o] < RankW[w,Husband[w]]
    => RankM[o,Wife[o]] < RankM[o,w];
 // stability 2:
 forall(m in Men, o in Women)
   RankM[m,o] < RankM[m,Wife[m]]
    => RankW[o,Husband[o]] < RankW[o,m]
};
```

Figure 5: Published OPL model of the original Stable Marriage problem

A First OPL **Model.** This problem was first modelled as a constraint program in the reference manual of ILOG SOLVER 4.0 (in 1997), and then modelled in OPL in a significantly simpler way in [14]: see Figure 5 for (a syntactic and re-typeset variant of) that model. The enumerated sets of women and men are to be read in at run-time, just like the two 2D arrays of rankings of the women and men for each other. Assertions could be added to ensure that both sets are of size n, and that each person's ranking is a permutation of the range $1{:}n$. The sought marriages are modelled in a redundant way, via two 1D arrays of variables representing the (unique) husband of each woman and the (unique) wife of each man, respectively. A linking constraint is thus necessary to ensure that both total functions are the inverse of each other, that is to achieve a bijection. To achieve better propagation, this linking constraint is here expressed for *both* functions, requiring every person to be identical to the spouse of their spouse. Finally, two stability constraints capture the definition of stability of a marriage. This model showcases a key

```
enum Women ...;
enum Men ...;
int RankW[Women,Men] = ...;
int RankM[Men,Women] = ...;
var Women Wife[Men];
solve {
 alldifferent(Wife);  // bijection
 // stability 1:
 forall(w in Women, m,o in Men)
   Wife[m]=w & RankW[w,o] < RankW[w,m]
    => RankM[o,Wife[o]] < RankM[o,w];
 // stability 2:
 forall(m,p in Men, o in Women)
   Wife[p]=o & RankM[m,o]<RankM[m,Wife[m]]
    => RankW[o,p] < RankW[o,m]
};
```

Figure 6: A second OPL model of the original Stable Marriage problem

feature of OPL, namely that expressions with variables, such as `Wife[m]`, can be used to index arrays (of variables). This feature is crucial for concise and clear modelling of many problems. Furthermore, the constraint-solving algorithm of OPL uses such expressions to reduce the domains of the variables in the arrays via bi-directional propagation.

A Second OPL **Model.** Let us design another OPL model, which less well exploits the features of the OPL solver. It non-redundantly models the marriages, namely by a single total function, that is a 1D array of variables representing the (unique) wife of each man, say. (Due to the symmetry in the definition of marriage stability, the actual choice probably does not matter. However, in an optimisation version of the problem, such as below, the choice *would* matter if there were some women or men that are more preferred than the others.) Figure 6 has the resulting model. There is no need for any linking constraints. To enforce the bijectiveness of the function, it suffices to constrain all variables to be different. Finally, the two stability constraints had to be rephrased slightly, as there is now no direct way of retrieving the husband of a given woman: this is achieved by iterating over all the men. This model is probably less efficient, and this has been the case with the instance data I tried.

A Third OPL **Model.** A third OPL model represents the marriages in a 2D array `Marriage` of 0/1 integer variables, indexed by the women and men, so that `Marriage[w,m]` is `1` iff woman `w` is married to man `m`. Figure 7 shows the resulting model. Two bijectiveness constraints are necessary to enforce that every person has exactly one spouse, that is that there is exactly one `1` in each row and in each column. The two stability constraints are obtained from those of the second model by the same technique as the stability constraints of the second model were obtained from those of the first one. This model is probably less efficient than the second one, and this has been the case with the instance data I tried.

```
enum Women ...;
enum Men ...;
int RankW[Women,Men] = ...;
int RankM[Men,Women] = ...;
var int Marriage[Women,Men] in 0..1;
solve {
 forall(w in Women)         // bijection 1
   sum(m in Men) Marriage[w,m] = 1;
 forall(m in Men)           // bijection 2
   sum(w in Women) Marriage[w,m] = 1;
 forall(w,p in Women, m,o in Men) {
   // stability 1:
   Marriage[w,m]=1 & Marriage[p,o]=1
    & RankW[w,o] < RankW[w,m]
       => RankM[o,p] < RankM[o,w];
   // stability 2:
   Marriage[w,m]=1 & Marriage[p,o]=1
    & RankM[m,p] < RankM[m,w]
       => RankW[p,o] < RankW[p,m];
 }
};
```

Figure 7: A third OPL model of the original Stable Marriage problem

```
enum Women, Men
nat RankW[Women,Men], RankM[Men,Women]
var Women 1:1 # 1:1 Men Marriage  // bij.
solve {
 forall(w#m, p#o in Marriage) {
   RankW[w,o] < RankW[w,m] // stability 1
    => RankM[o,p] < RankM[o,w]
   &
   RankM[m,p] < RankM[m,w] // stability 2
    => RankW[p,o] < RankW[p,m]
 }
}
```

Figure 8: An ESRA model of the original Stable Marriage problem

An ESRA Model. Figure 8 shows an ESRA model of the problem. The modelling of the instance data is similar to the OPL models above. The sought marriages are modelled as a relation over the sets of women and men, such that it is a bijection: for every woman, there is exactly one man, and for every man, there is exactly one woman. There is no need for any linking constraints. The two stability constraints directly capture the definition of stability of a marriage, by iterating over all pairs of married couples.

This model can be translated into various OPL models, not unlike the ones above, using the (possibly redundant) ways of representing relations, and exploiting insights gained from thorough studies of bijections [12; 15]. The advantages of abstract relational modelling are that the modeller need not worry about the actual (and possibly redundant) representation of the relation and its multiplicity constraints, and that the model is (thus) more flexible when the problem changes, as shown next, as well as in Sections 4.2 and 4.3.

Model Maintenance. Paraphrasing the two model flexibility challenges raised in Section 8.4 of [10], where a different bijection problem was examined, let us now consider the following two additional constraints for the problem.

First, suppose that (for whatever reason) we cannot have both ⟨Sue, Jim⟩ and ⟨Pat, Bob⟩ as married couples. In the first and second OPL models, this can easily be expressed as the following new constraint:

```
not(Wife[Jim]=Sue & Wife[Bob]=Pat)
```

or, probably more efficiently but at a lower level, as:

```
(Wife[Jim]=Sue) + (Wife[Bob]=Pat) <= 1
```

whereas in the third OPL model, this can also easily be expressed, say as:

```
not(Marriage[Sue,Jim]=1
  & Marriage[Pat,Bob]=1)
```

or, probably more efficiently but less naturally, as:

```
Marriage[Sue,Jim]+Marriage[Pat,Bob] <= 1
```

In the ESRA model, the constraint can also easily be expressed, say as follows:

```
not(Sue#Jim in Marriage
  & Pat#Bob in Marriage)
```

In other words, both OPL and ESRA easily handle this challenge. However, in a lower-level language than OPL, the equivalents of the first and second OPL models make this new constraint rather hard to express (see [10] for a solution).

Second, suppose that (for whatever reason) Bob must marry a woman coming after the wife of Jim in the enumeration of women. In the first and second OPL models, this can easily be expressed as the following new constraint:

```
ord(Wife[Bob]) > ord(Wife[Jim])
```

but in the third OPL model, this can be expressed only with difficulty, say as:

```
sum(w in Women) Marriage[w,Bob] * ord(w)
> sum(w in Women) Marriage[w,Jim] *ord(w)
```

In the ESRA model, the constraint can easily be expressed, say as follows:

```
ord(Bob.~Marriage) > ord(Jim.~Marriage)
```

In other words, this time *some* OPL models are *not* easy to maintain, so only ESRA easily handles the challenge, because its model is unencumbered with representation details. Worse, in a lower-level language than OPL, the equivalent of the third OPL model makes this additional constraint even harder to express (see [10] for a solution).

Model maintenance at the high ESRA level thus reduces to adapting to the new problem, with all representation (and thus solving) issues left to the compiler. At lower levels, even with OPL, model maintenance may become quite tedious, because the already made representation choices have to be taken into account and because the lower-level notation is more awkward. Worse, a representation change, a redundancy elimination, or a redundancy introduction (such as a model integration or the addition of implied constraints) may "have to" be operated, because it is unlikely that, for the considered training instances or even in general, the 'best' representation is the same for bijections as for full relations, say. Such model maintenance may become necessary not only when the actual problem changes, but also because the distribution of instances on which the model is deployed becomes different from the training distribution used when the model was formulated. But the modeller may be unwilling or unable to do this experimentation for finding the 'best' model, or s/he may be unaware of insights gained from a general empirical study, such as on how to 'best' model bijections [12].

Fortunately, relations and their particular cases (such as partial functions, total functions, injections, surjections, bijections, and so on) are a *single*, powerful concept for elegantly modelling many aspects of combinatorial optimisation problems. Also, there are *not too many* different, and even *standard*, ways of representing relations and relational expressions. Therefore, I advocate that the compiler can actually make a (systematic) empirical evaluation of candidate representations, using (real-life) training instances of the problem. In the absence of such training instances, such a compiler would simply be non-deterministic. Also, theoretical studies such as [15] should be made for particular cases of relations in order to obtain rules stating when a representation is advisable and when not, thereby reducing the volume of such empirical studies by compilers.

4.2 Monogamy/Monoandry Version

Let us change the Stable Marriage problem to make it more realistic. The numbers of women and men may now differ, so it may be impossible to arrange marriages for everyone. However, let us keep the previous constraint that everyone may marry at most one person, of the opposite sex, hence we call this the *monogamy/monoandry version*. The sought marriages no longer form a bijection, but rather *partial functions* between the sets of women and men. The definition of marriage stability now has to be modified. A monogamic/monoandric marriage is said to be *stable* if, whenever spouse s prefers some other partner, this partner *is* married, *and* s/he prefers her/his spouse to s. To prevent the problem from now having the empty set of marriages as a solution, it has to become an optimisation problem. Various 'happiness' functions on the marriages can be defined, but I here just take a simple one and maximise their number.

Figure 9 shows an ESRA model of this new problem.[6] A cost function appeared, the lower bounds of the multiplicities were changed from `1` to `0`, and the stability constraints were rephrased to reflect the new definition. (The same stability constraints could actually also have been used in the model of Figure 8, because the new definition of stability implies the original one in its context.) Model maintenance was thus easy, because no representation issues had to be dealt with.

4.3 Polyandry Version

Imagine now a country where the law allows women to marry up to 3 men, but men may marry only 1 woman. Also consider that all women who signed up at the agency need to marry (for some reason). We call this the *polyandry version*.

[6] OPL does *not* have existential quantification.

```
enum Women, Men
nat RankW[Women,Men], RankM[Men,Women]
var Women 0:1 # 0:1 Men Marriage
maximise
 count(w#m in Marriage)
subject to {
 forall(w#m in Marriage) {
   forall(o in Men)        // stability 1
     RankW[w,o] < RankW[w,m] =>
      exists(p in Women: p#o in Marriage)
        RankM[o,p] < RankM[o,w]
   &
   forall(p in Women)      // stability 2
     RankM[m,p] < RankM[m,w] =>
      exists(o in Men: p#o in Marriage)
        RankW[p,o] < RankW[p,m]
 }
}
```

Figure 9: An ESRA model of the monogamic/monoandric Stable Marriage problem

The sought marriages now form a full *relation* between the sets of women and men. The definition of marriage stability again has to be modified. A polyandric marriage is said to be *stable* if, whenever spouse s prefers some other partner, this partner *is* married, *and* s/he prefers *all* her/his spouses to s. If at least as many men as women have signed up at the agency, the problem again becomes a decision problem.

Figure 10 shows an ESRA model of this new problem.[7] The cost function disappeared, the multiplicities were changed, and the second stability constraint was rephrased to reflect the new definition, as the final `exists` became a `forall`. (The same stability constraints could actually also have been used in the model of Figure 8, because the new definition of stability implies the original one in its context.) Again, the model maintenance was unburdened by representation issues.

5 Conclusion

Summary. I have argued that a high-level abstract constraint modelling language would enable an automatable empirical determination — by a compiler — of how to 'best' represent the variables of a combinatorial optimisation problem. This compilation process would be based on (the few) standard ways of (possibly redundantly) representing high-level concepts such as relations, on theoretically or empirically gained knowledge about when to deploy which representation, as well as on (real-life) training instances of the problem. In the absence of training instances, such a compiler would simply be non-deterministic.

The advantage of such high-level modelling is that model formulation and maintenance become much easier, because unencumbered by early if not uninformed commitments to representation choices. Automatable model reformulation also becomes possible [8].

[7] OPL does *not* allow problem variables in the filtering formula after the 'such that' (denoted '`:`') of universal quantifications.

```
enum Women, Men
nat RankW[Women,Men], RankM[Men,Women]
var Women 1:3 # 0:1 Men Marriage
solve {
 forall(w#m in Marriage) {
   // stability 1:
   forall(o in Men)        // stability 1
     RankW[w,o] < RankW[w,m] =>
      exists(p in Women: p#o in Marriage)
        RankM[o,p] < RankM[o,w]
   &
   // stability 2:
   forall(p in Women)      // stability 2
     RankM[m,p] < RankM[m,w] =>
      forall(o in Men: p#o in Marriage)
        RankW[p,o] < RankW[p,m]
 }
}
```

Figure 10: An ESRA model of the polyandric Stable Marriage problem

I have shown that a suitable first-order relational calculus with sets is a good candidate for such a modelling language, as it gives rise to very natural and easy-to-maintain models of combinatorial optimisation problems.

Related Work. This research owes a lot to previous work on relational modelling in formal methods and on ERA-style semantic data modelling, especially to the ALLOY object modelling language [9], which itself gained much from the Z specification notation [13] (and learned from UML/OCL how not to do it). Contrary to ERA modelling, I do not distinguish between attributes and relations.

In constraint programming, OPL [14] stands out as a medium-level constraint modelling language, and ALMA [2] is also becoming a very powerful notation, on top of MODULA-2. Our ESRA shares with them the quest for a practical declarative modelling language based on a strongly-typed (full) first-order logic with arrays (and with the look of an imperative language), while dispensing with such hard-to-properly-implement and rarely-necessary (for constraint modelling) 'luxuries' as recursion and unbounded quantification. As shown, ESRA even goes beyond them, by advocating an abstract view of sets and relations.

Future Work. I plan to evolve the ESRA report [3] into a complete (and more formal) description of the concepts, syntax, and semantics of the full ESRA language, as well as to implement its semantics with an ESRA-to-OPL compiler. I can then tackle the joint consideration of the modelling and the solver parameterisation, as well as automated reformulation at the ESRA level. The language could be extended with a syntax for declaratively expressing symmetries [6], so that the compiler can generate suitable code for breaking them. Also, a graphical language could be developed for the variable modelling, including the multiplicity constraints on relations, so that only the cost function and the constraints would need to be textually expressed.

Acknowledgements. This research is partly funded under grant 221-99-369 of TFR, the Swedish Research Council for Engineering Sciences. I would also like to acknowledge fruitful discussions with my students Zeynep Kızıltan and Brahim Hnich, as well as the useful feedback from the referee.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. ALMA-0: An imperative language that supports declarative programming. *ACM TOPLAS* 20(5):1014–1066, 1998.

[3] P. Flener and B. Hnich. The syntax and semantics of ESRA. Evolving internal report of the ASTRA Team, at http://www.dis.uu.se/∼pierref/astra/.

[4] P. Flener, B. Hnich, and Z. Kızıltan. Compiling high-level type constructors in constraint programming. In: I.V. Ramakrishnan (ed), *Proc. of PADL'01*, pp. 229–244. LNCS 1990. Springer-Verlag, 2001.

[5] P. Flener, B. Hnich, and Z. Kızıltan. A meta-heuristic for subset problems. In: I.V. Ramakrishnan (ed), *Proc. of PADL'01*, pp. 274–287. LNCS 1990. Springer-Verlag, 2001.

[6] I.P. Gent and B.M. Smith. Symmetry breaking during search in constraint programming. *Proc. of ECAI'00*, pp. 599–603. John Wiley & Sons, 2000.

[7] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.

[8] B. Hnich and P. Flener. High-level reformulation of constraint programs. In: Ph. Codognet (ed), *Proc. of JF-PLC'01*. Éditions Hermès, 2001.

[9] D. Jackson. ALLOY: A lightweight object modelling notation. *ACM TOPLAS*, forthcoming. Available from http://sdg.lcs.mit.edu/∼dnj/.

[10] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

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

[12] B.M. Smith. Modelling a permutation problem. RR 18, University of Leeds (United Kingdom), School of Computer Studies, 2000. Available from http://www.comp.leeds.ac.uk/bms/.

[13] J.M. Spivey. *The Z Notation: A Reference Manual* (second edition). Prentice-Hall, 1992.

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

[15] T. Walsh. Permutation problems and channelling constraints. TR 26, APES Group, 2001. http://www.dcs.st-and.ac.uk/∼apes/.