Program Synthesis for Combinatorial Optimisation Problems Position Statement *

Pierre Flener
Department of Information Technology
Uppsala University, Box 337, S – 751 05 Uppsala, Sweden
pierref@csd.uu.se ASTRA group: http://www.csd.uu.se/∼pierref/astra/

Abstract

A high-level abstract-datatype-based constraint modelling language opens the door to an automatable empirical determination — by a synthesiser — of how to 'best' represent the decision 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 synthesiser would simply be non-deterministic. A first-order relational calculus is a good candidate for such a language, as it gives rise to very natural and easy-to-maintain models of combinatorial optimisation problems.

Introduction

Combinatorial optimisation problems are increasingly ubiquitous and crucial in industry. Indeed, staying competitive in the global New Economy requires the efficient modelling and solving of such problems, whose instances are getting larger and harder. Examples are production planning subject to customer demand and resource availability so that sales are maximised, and air traffic control subject to safety protocols so that flight times are minimised. Appropriate values for the decision variables have to be found within their domains, subject to some constraints, such that some optional objective function on these variables takes an optimal value.

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 [10] specification languages, the ALLOY [6] 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. We examine whether constraint modelling can benefit from these 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 [5]. 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, which consumes more memory but is able to support more set expressions and constraints. Both representations 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 arrays. Indeed, a total function f can be represented as a 1-d array of variables over the range of f, indexed by its domain, or as a 2-d array of 0/1 variables, indexed by the domain *and* range of f, or even with some redundancy, as long as channelling constraints relate the parts of the redundant representation. Other than retrieving the (unique) image under a total function of a domain element, there has been no support for relational expressions.

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

We 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 synthesisers [7,8]. We thus here only aim at techniques that find the 'best' model for a *given* solver, under its *default* settings.

Relational Modelling with ESRA

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.

The key design decisions for our constraint modelling language — called ESRA — are as follows. We want to capture common modelling idioms in abstract datatypes, especially for 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 num-

*These ideas are the result of many fruitful discussions with Brahim Hnich and Zeynep Kızıltan. This research is partly funded under grant 221-99-369 of VR, the Swedish Research Council, and under grant IG2001-67 of STINT, the Swedish Foundation for International Cooperation in Research and Higher Education.

ber of combinatorial optimisation problems. We (currently) do not support procedures, and hence no procedure calls and no recursion. Similarly, 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 a 'lower-128 ASCII' syntax, unlike the LaTeX-requiring syntax of Z, as well as a JAVA-style declaration of the universally quantified variables. For reasons of space, we here only introduce the concepts of ESRA that are actually illustrated in this paper. Also, we can "only" give an informal semantics. The reader may monitor www.csd.uu.se/∼pierref/astra for a complete description of the full language.

Modelling the Data. 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. 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.

Relations are declared using the # 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. 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. We thus (currently) restrict the focus to *binary* relations, between primitive types only. 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 a specific syntax, we prefer keeping the notation lean and leave their specialised handling to the synthesiser. This has the further advantage that only the multiplicities need to be changed during model maintenance, say when a function becomes a relation.

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

Modelling the Cost Function and the Constraints. Expressions are constructed in the usual way. The usual arithmetic operators are available, such as `card` for the cardinality of a set expression, `ord` for the position of an identifier in an enumeration, and `sum` for the sum of a bounded (and possibly filtered) number of numeric expressions. 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`. The elements of a relation are represented as `a#b` pairs.

First-order logic formulas are also constructed in the usual way. 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, association, and precedence rules apply.

The *cost function* is a numeric expression that has to be either minimised or maximised. The *constraints* on the decision variables are a conjunction of formulas.

The Warehouse Location Problem

A company considers opening warehouses on some candidate locations 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 warehouse (C_1). Each candidate warehouse has a capacity designating the maximum number of stores 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* from the set of stores into the set of warehouses, and the set of warehouses to be opened is the *range* of that function.

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 [11]. There, the sought total function is modelled by a 1-d array of variables representing the (unique) warehouse that supplies each store, thereby capturing the 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 1-d array `OW` of 0/1 variables, such that `OW[w]` is `1` iff warehouse `w` is opened. A channelling constraint is then necessary, expressing that a warehouse that is actually supplying some store must be opened. The cost function and constraint C_2 can only be expressed in a low-level way, namely by reinterpreting the Booleans of `OW` and the truth values of local constraints as numeric weights. Things become even more awkward if we non-redundantly model the supply function, namely just by the 1-d array of variables representing the warehouse that supplies each store. On the instance data we tried, this model is actually an order of magnitude more efficient (by all measures) than the published one, but it is much less readable. This shows that redundancy elimination may pay off in performance, but it may just as well be re-

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

Figure 1: The Warehouse Location problem

	Hal	Jim	Bob
Nat	1	2	3
Eve	2	3	1
Pat	3	2	1

	Nat	Eve	Pat
Hal	3	1	2
Jim	3	1	2
Bob	3	2	1

Figure 2: Rankings of the women for the men (left), and rankings of the men for the women (right)

dundancy introduction. But this is hard to guess, as human intuition may be weak here.

Figure 1 shows an ESRA model of the problem. 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. From this model, lower-level models can be synthesised, including the ones discussed above.

The Stable Marriage Problem

Original Version. Consider a dating agency where an equal number n of women and men have signed up and are willing to marry any opposite-sex person of the group. They have ranked all possible spouses by decreasing preference. Figure 2 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 *stable* if, whenever spouse s prefers some other partner, this partner prefers her/his spouse to s. So 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.

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 [11]. The marriages are modelled in a redundant way, via two 1-d arrays of variables representing the (unique) husband of each woman and the (unique) wife of each man, respectively. A channelling constraint is necessary to ensure that both total functions are the inverse of each other, that is to achieve a bijection. To achieve better propagation, this channelling

```
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 3: The original Stable Marriage problem

constraint is expressed for *both* functions, requiring every person to be identical to the spouse of their spouse.

A second model would non-redundantly model the marriages, namely by a single total function, that is a 1-d array of variables representing the wife of each man, say. To enforce the bijectiveness of this function, all variables are constrained to be different. This model is probably less efficient, and this has been the case with the instance data we tried.

A third model would model the marriages in a 2-d 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`. Two bijectiveness constraints are necessary to enforce that every person has exactly one spouse, so that there is exactly one `1` in each row and in each column. This model is probably less efficient than the second one, and this has been the case with the instance data we tried.

Figure 3 shows an ESRA model of the problem. The marriages are modelled as a relation over the sets of women and men, such that it is a bijection. From this model, lower-level models can be synthesised, including the ones above, using the various ways of representing relations, and exploiting insights gained from thorough studies of bijections [9,12].

Model Maintenance. Relations and their particular cases (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, we advocate that the synthesiser 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 synthesiser would simply be non-deterministic. Also, theoretical studies such as [12] 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 synthesisers.

Model maintenance at the high ESRA level reduces to adapting to the new problem and re-synthesising, as all representation (and thus solving) issues are left to the synthesiser. At lower levels, model maintenance is quite tedious, as the early if not uninformed representation choices have to be taken into account and as the lower-level notation is more awkward. Worse, a representation change, a redundancy elimination, or a redundancy introduction (such as

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

Figure 4: The Polyandric Stable Marriage problem

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 in general, the 'best' representation is the same for bijections as for full relations, say.

Such re-synthesis is also necessary when 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 [9].

Polyandry Version. Imagine 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. The sought marriages now form a full *relation* between the sets of women and men. A polyandric marriage is *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 remains a decision problem.

Figure 4 shows an ESRA model of this new problem. The multiplicities were changed, 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 3, because the new definition of stability implies the original one in its context.) The model maintenance was indeed unburdened by representation issues.

Conclusion

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 [6], which itself gained much from the Z specification notation [10] (and learned from UML/OCL how not to do it). Contrary to ERA modelling, we do not distinguish between attributes and relations.

In constraint programming, OPL [11] 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 language 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 relations.

Current and Future Work. Our ESRA language is an extension of a streamlined (significant) subset of OPL. A prototype ESRA-to-OPL synthesiser [4] has been implemented by Simon Wrang. The semantics 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. We can then tackle the joint consideration of the modelling and the solver parameterisation. The synthesiser will also benefit from our work on symmetry-reducing/breaking constraints [3]. A graphical language can be developed for the variable modelling, including the multiplicity constraints on relations, so that only the cost function and the other constraints need to be textually expressed.

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. An imperative language that supports declarative programming. *ACM TOPLAS* 20(5):1014–1066, 1998.
3. P. Flener, A. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, J. Pearson, and T. Walsh. Symmetry in matrix models. In *CP'01 Workshop on Symmetry in Constraints*.
4. P. Flener, B. Hnich, and Z. Kızıltan. Compiling high-level type constructors in constraint programming. In *Proc. of PADL'01*. LNCS 1990. Springer-Verlag, 2001.
5. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.
6. D. Jackson. ALLOY: A lightweight object modelling notation. *ACM TOPLAS*, forthcoming.
7. Z. Kızıltan, P. Flener, and B. Hnich. Towards inferring labelling heuristics for CSP application domains. In *Proc. of KI'01*. LNAI 2174. Springer-Verlag, 2001.
8. S. Minton. Automatically configuring constraint satisfaction programs. *Constraints* 1(1–2):7–43, 1996.
9. B.M. Smith. Modelling a permutation problem. RR 18, Univ. of Leeds (UK), School of Computer Studies, 2000.
10. J.M. Spivey. *The* Z *Notation: A Reference Manual* (second edition). Prentice-Hall, 1992.
11. P. Van Hentenryck. *The* OPL *Optimization Programming Language*. The MIT Press, 1999.
12. T. Walsh. Permutation problems and channelling constraints. TR 26 at www.dcs.st-and.ac.uk/∼apes, 2001.