Introducing ESRA, a Relational Language for Modelling Combinatorial Problems [⋆]

Pierre Flener, Justin Pearson, and Magnus Ågren [⋆⋆]

Department of Information Technology
Uppsala University, Box 337, S – 751 05 Uppsala, Sweden
{pierref,justin,agren}@it.uu.se

Abstract. Current-generation constraint programming languages are considered by many, especially in industry, to be too low-level, difficult, and large. We argue that solver-independent, high-level relational constraint modelling leads to a simpler and smaller language, to more concise, intuitive, and analysable models, as well as to more efficient and effective model formulation, maintenance, reformulation, and verification. All this can be achieved without sacrificing the possibility of efficient solving, so that even time-pressed or less competent modellers can be well assisted. Towards this, we propose the ESRA relational constraint modelling language, showcase its elegance on some well-known problems, and outline a compilation philosophy for such languages.

1 Introduction

Current-generation constraint programming languages are considered by many, especially in industry, to be too low-level, difficult, and large. Consequently, their solvers are not in as widespread use as they ought to be, and constraint programming is still fairly unknown in many application domains, such as molecular biology. In order to unleash the proven powers of constraint technology and make it available to a wider range of problem modellers, a solver-independent, higher-level, simpler, and smaller modelling notation is needed.

In our opinion, even recent commercial languages such as OPL [31] do not go far enough in that direction. Many common modelling patterns have not been captured in special constructs. They have to be painstakingly spelled out each time, at a high risk for errors, often using low-level devices such as reification.

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 [29] specification languages, the ALLOY [16] object modelling language, and the *Object Constraint Language* (OCL) [35] of the *Unified Modelling Language* (UML) [27]. In semantic data modelling this had been long advocated; most notably via entity-relationship-attribute (ERA) diagrams.

[⋆] A previous version of this paper appears pages 63–77 in the informally published proceedings of the *Second International Workshop Modelling and Reformulating CSPs*, available at `http://www-users.cs.york.ac.uk/~frisch/Reformulation/03/`.

[⋆⋆] The authors' names are ordered according to the Swedish alphabet.

Sets and set expressions started appearing as modelling devices in some constraint languages. Set variables are often implemented by the set interval representation [13]. In the absence of such an explicit set concept, modellers usually painstakingly represent a set variable by its characteristic function, namely as a sequence of 0/1 integer variables, as long as the size of the domain of the set.

Relations have not received much attention yet in constraint programming languages, except total functions, via arrays. Indeed, a total function f can be represented in many ways [15], say as a 1-dimensional array of variables over the range of f, indexed by its domain, or as a 2-dimensional array of Boolean variables, indexed by the domain and range of f, or as a 1-dimensional array of set variables over the domain of f, indexed by its range, or even with some redundancy. Other than retrieving the (unique) image under a total function of a domain element, there has been no support for relational expressions.

Matrix modelling [8, 10, 31] has been advocated as one way of capturing common modelling patterns. Alternatively, it has been argued [11, 15] that functions, and hence relations, should be supported by an abstract datatype (ADT). It is then *the compiler* that must (help the modeller) choose a suitable representation, say in a contemporary constraint programming language, for each instance of the ADT, using empirically or theoretically gained modelling insights.

We here demonstrate, as originally conjectured in [9], that a suitable first-order relational calculus is a good basis for a high-level, ADT-based, and solver-independent constraint modelling language. It gives rise to very natural and easy-to-maintain models of combinatorial problems. Even in the (temporary) absence of a corresponding high-level search language, this generality does not necessarily come at a loss in solving efficiency, as abstract relational models are devoid of representation details so that the results of analysis can be exploited.

Our aims here are only to justify and present our new language, called ESRA, to illustrate its elegance and the flexibility of its models by some examples, and to argue that it can be compiled into efficient models in lower-level (constraint programming) languages. The syntax, denotational semantics, and type system of the proposed language are discussed in full detail in an online appendix [12] and a second prototype of the advocated compiler is under development.

The rest of this paper is organised as follows. In Section 2, we present our relational language for modelling combinatorial problems and deploy it on three real-life problems before discussing its compilation. This allows us to list, in Section 3, the benefits of relational modelling. Finally, in Section 4, we conclude as well as discuss related and future work.

2 Relational Constraint Modelling with ESRA

In Section 2.1, we justify the design decisions behind our new ESRA constraint modelling language, targeted at constraint programmers. Then, in Section 2.2, we introduce its concepts, syntax, type system, and semantics. Next, in Section 2.3, we deploy ESRA on three real-life problems. Finally, in Section 2.4, we discuss the design of our prototype compilers for ESRA.

2.1 Design Decisions

The key design decisions for our new relational constraint modelling language — called ESRA for *Executable Symbolism for Relational Algebra* — were as follows.

We want to capture common modelling idioms in a new abstract datatype for relations, so as to design a high-level and simple language. The constructs of the language are orthogonal, so as to keep the language small. Computational completeness is not aimed at, as long as the language is useful for elegantly modelling a large number of combinatorial problems.

We focus on *finite*, discrete domains. Relations are built from such domains and sets are viewed as unary relations. Theoretical difficulties are sidestepped by supporting only bounded quantification, but not negation nor sets of sets.

The language has an ASCII syntax, mimicking mathematical and logical notation as closely as possible, as well as a LaTeX-based syntax, especially used for pretty-printing models in that notation.

2.2 Concepts, Syntax, Type System, and Semantics of ESRA

For reasons of space, we only give an informal semantics. The interested reader is invited to consult [12] for a complete description of the language. Essentially, the semantics of the language is a conservative extension of existential second-order logic. Existential quantification of relations is used to assert that relations are to be found that satisfy sets of first-order constraints. This is in contrast with extensions of logic programming [6, 25] where second-order relations can be specified recursively using Horn clauses, which needs a much more careful treatment of the fixed-point semantics.

Code excerpts are here provided out of the semantic context of any particular problem statement, just to illustrate the syntax, but a suggested reading in plain English is always provided. In Section 2.3, we will actually start from plain English problem statements and show how they can be modelled in ESRA. Code excerpts are always given in the pretty-printed form, but we indicate the ASCII notation for every symbol where it necessarily differs.

An ESRA model starts with a sequence of declarations of named *domains* (or types) as well as named *constants* and *decision variables* that are tied to domains. Then comes the *objective*, which is to find values for the decision variables within their domains so that some *constraints* are satisfied and possibly some *cost expression* takes an optimal value.

The Type System. A *primitive domain* is a finite, extensionally given set of new names or integers, comma-separated and enclosed as usual in curly braces. An integer domain can also be given intensionally as a finite integer interval, by separating its lower and upper bounds with '$\ldots$' (denoted in ASCII by '`..`'), without using curly braces. When these bounds coincide, the corresponding singleton domain $n \ldots n$ or $\{n\}$ can be abbreviated to n. Context always determines whether an integer n designates itself or the singleton domain $\{n\}$. A domain can also be given intensionally using set comprehension notation.

The only *predefined* primitive domains are the sets $\mathbb{N}$ (denoted in ASCII by '`nat`') and $\mathbb{Z}$ (denoted in ASCII by '`int`'), which are '$0 \ldots \text{sup}$' and '$\text{inf} \ldots \text{sup}$' respectively, where the predefined constant identifiers 'inf' and 'sup' stand for the smallest negative and largest positive representable integers respectively. *User-defined* primitive domains are declared after the 'dom' keyword and initialised at compile-time, using the '=' symbol, or at run-time, via a datafile, otherwise interactively.

Example 1. The statement

$$\text{dom } \mathit{Varieties}, \mathit{Blocks}$$

declares two domains called *Varieties* and *Blocks* that are to be initialised at run-time. As in OPL [31], this neatly separates the problem model from its instance data, so that the actual constraint satisfaction problem is obtained at run-time.

Similarly, the statement

$$\text{dom } \mathit{Players} = 1 \ldots g * s,\ \ \mathit{Weeks} = 1 \ldots w,\ \ \mathit{Groups} = 1 \ldots g$$

where g, s, w are integer-constant identifiers (assumed previously declared, in a way shown below), declares integer domains called *Players*, *Weeks*, and *Groups* that are initialised at compile-time.

Finally, the declaration

$$\text{dom } \mathit{Even} = \{i \mid i : 0 \ldots 100 \mid i \,\%\, 2 = 0\}$$

initialises the domain *Even* of all even natural numbers up to 100.

The usual binary infix $\times$ constructor (denoted in ASCII by '`#`') allows the construction of Cartesian products.

The only *constructed domains* are relational domains. In order to simultaneously capture frequently occurring multiplicity constraints on relations, we offer a parameterised binary infix $\times$ domain constructor. The relational domain $A\ {}^{M_1}\!\!\times^{M_2} B$, where A and B are (possibly Cartesian products of) primitive domains, designates a set of binary relations in $A \times B$. The optional M_1 and M_2, called *multiplicities*, must be integer sets and have the following semantics: for every element a of A, the number of elements of B related to a must be in M_1, while for every element b of B, the number of elements of A related to b must be in M_2.[1] An omitted multiplicity stands for $\mathbb{N}$.

Example 2. The constructed domain

$$\mathit{Varieties}\ {}^{r}\!\!\times^{k}\ \mathit{Blocks}$$

designates the set of all relations in $\mathit{Varieties} \times \mathit{Blocks}$ where every variety occurs in exactly r blocks and every block contains exactly k varieties. These are two occurrences where an integer abbreviates the singleton domain containing it.

[1] Note that our syntax is the opposite of the UML one, say, where the multiplicities are written in the other order, with the *same* semantics. That convention can however *not* be usefully upgraded to Cartesian products of arity higher than 2.

In the absence of such facilities for relations and their multiplicities, a relational domain would have to be modelled using arrays, say. This may be a premature commitment to a concrete data structure, as the modeller may not know yet, especially prior to experimentation, which particular (array-based) representation of a relational decision variable will lead to the most efficient solving. The problem constraints, including the multiplicities, would have to be formulated in the constraints part of the model, based on the chosen representation. If the experiments revealed that another representation should be tried, then the modeller would have to first painstakingly reformulate the declaration of the decision variable as well as all its constraints. Our ADT view of relations overcomes this flaw: it is now *the compiler* that must (help the modeller) choose a suitable representation for each instance of the ADT by using empirically or theoretically gained insights. Also, multiplicities need not become counting constraints, but are succinctly and conveniently captured in the declaration.

We view sets as unary relations: $A\ M$, where A is a domain and M an integer set, constructs the domain of all subsets of A whose cardinality is in M. The multiplicity M is mandatory here; otherwise there would be ambiguity whether a value of the domain A is an element or an arbitrarily sized subset of A.

For total and partial functions, the left-hand multiplicity M_1 is $1 \dots 1$ and $0 \dots 1$ respectively. In order to dispense with these left-hand multiplicities for total and partial functions, we offer the usual $\longrightarrow$ and $\not\longrightarrow$ (denoted in ASCII by '`->`' and '`+>`') domain constructors respectively, as shorthands. They may still have right-hand multiplicities though.

For injections, surjections, and bijections, the right-hand multiplicity M_2 is $0 \dots 1$, $1 \dots \mathrm{sup}$, and $1 \dots 1$ respectively. Rather than elevating these particular cases of functions to first-class concepts with an invented specific syntax in ESRA, we prefer keeping our language lean and close to mathematical notation.

Example 3. The constructed domain

$$(\mathit{Players} \times \mathit{Weeks}) \longrightarrow^{s*w} \mathit{Groups}$$

designates the set of all total functions from $\mathit{Players} \times \mathit{Weeks}$ into Groups such that every group is related to exactly sw (player,week) pairs.

We provide no support (yet) for bags and sequences, as relations provide enough challenges for the time being. Note that a *bag* can be modelled as a total function from its domain into $\mathbb{N}$, giving the repetition count of each element. Similarly, a *sequence* of length n can be modelled as a total function from $1 \dots n$ into its domain, telling which element is at each position. This does *not* mean that the representation of bags and sequences is fixed (to the one of total functions), because, as we shall see in Section 2.4, the various relations (and thus total functions) of a model need not have the same representation.

Modelling the Instance Data and Decision Variables. All identifier declarations are strongly typed and denote variables that are implicitly universally quantified over the entire model, with the constants expected to be ground before search begins while the decision variables can still be unbound at that moment.

Like the user-defined primitive domains, constants help describe the instance data of a problem. A constant identifier is declared after the 'cst' keyword and is tied to its domain by ':', meaning set membership. Constants are initialised at compile-time, using the '=' symbol, or at run-time, via a datafile, otherwise interactively. Again, run-time initialisation provides a neat separation of problem models and problem instances.

Example 4. The statement

$$\text{cst } r, k, \lambda : \mathbb{N}$$

declares three natural number constants that are to be initialised at run-time.

As already seen in Examples 2 and 3, the availability of total functions makes arrays unnecessary. The statement

$$\text{cst } \mathit{CrewSize} : \mathit{Guests} \longrightarrow \mathbb{N},\ \ \mathit{SpareCap} : \mathit{Hosts} \longrightarrow \mathbb{N}$$

declares two natural-number functions, to be provided at run-time.

A decision-variable identifier is declared after the 'var' keyword and is tied to its domain by ':'.

Example 5. The statement

$$\text{var } \mathit{BIBD} : \mathit{Varieties}\ {}^{r}\!\times^{k}\ \mathit{Blocks}$$

declares a relation called *BIBD* of the domain of Example 2.

Modelling the Cost Expression and the Constraints. *Expressions* and first-order logic *formulas* are constructed in the usual way.

For *numeric expressions*, the arguments are either integers or identifiers of the domain $\mathbb{N}$ or $\mathbb{Z}$, including the predefined constants 'inf' and 'sup'. Usual unary ($-$, 'abs' for absolute value, and 'card' for the cardinality of a set expression), binary infix ($+$, $-$, $*$, $/$ for integer quotient, and $\%$ for integer remainder), and aggregate ($\sum$, denoted in ASCII by '`sum`') arithmetic operators are available. A sum is indexed by local variables ranging over finite sets, which may be filtered on-the-fly by a condition given after the '$|$' symbol (read 'such that').

Sets obey the same rules as domains. So, for *set expressions*, the arguments are either set identifiers or (intensionally or extensionally) given sets, including the predefined sets $\mathbb{N}$ and $\mathbb{Z}$. Only the (unparameterised) binary infix domain constructor $\times$ and its specialisations $\longrightarrow$ and $\not\longrightarrow$ are available as operators.

Finally *function expressions* are built by applying a function identifier to an argument tuple. We have found no use yet for any other operators on functions (but see the discussion of future work in Section 4).

Example 6. The numeric expression

$$\sum_{g:Guests \;|\; Schedule(g,p)=h} \mathit{CrewSize}(g)$$

denotes the sum of the crew sizes of all the guest boats that are scheduled to visit host h at period p, assuming this expression is within the scope of the local variables h and p. The nested function expression $\mathit{CrewSize}(g)$ stands for the size of the crew of guest g, which is a natural number according to Example 4.

Atoms are built from numeric expressions with the usual comparison predicates, such as the binary infix $=$, $\neq$, and $\leq$ (denoted in ASCII by '`=`', '`!=`', and '`=<`' respectively). Atoms also include the predefined 'true' and 'false', as well as references to the elements of a relation. We have found no use yet for any other predicates. Note that '$\in$' is unnecessary as $x \in S$ is equivalent to $S(x)$.

Example 7. The atom $\mathit{BIBD}(v_1, i)$ stands for the truth value of variety v_1 being related to block i in the *BIBD* relation of Example 5.

Formulas are built from atoms. The usual binary infix connectives ($\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, and $\Leftrightarrow$, denoted in ASCII by '`/\`', '`\/`', '`=>`', '`<=`', and '`<=>`' respectively) and quantifiers ($\forall$ and $\exists$, denoted in ASCII by '`forall`' and '`exists`' respectively) are available. A quantified formula is indexed by local variables ranging over finite sets, which may be filtered on-the-fly by a condition given after the '$|$' symbol (read 'such that'). As we provide a rich (enough) set of predicates, we are only interested in models that can be formulated positively, and thus dispense with the negation connective. The usual typing and precedence rules for operators and connectives apply. All binary operators associate to the left.

Example 8. The formula

$$\forall(p : \mathit{Periods},\ h : \mathit{Hosts}) \left(\sum_{g:Guests \;|\; Schedule(g,p)=h} \mathit{CrewSize}(g) \right) \leq \mathit{SpareCap}(h)$$

constrains the spare capacity of any host boat h not to be exceeded at any period p by the sum of the crew sizes of all the guest boats that are scheduled to visit host h at period p.

A generalisation of the $\exists$ quantifier turns out to be very useful. We define

$$\mathrm{count}(\mathit{Multiplicity})(x : \mathit{Set} \mid \mathit{Condition})$$

to hold if and only if the cardinality of the set comprehension $\{x : \mathit{Set} \mid \mathit{Condition}\}$ is in the integer set *Multiplicity*. So

$$\exists(x : \mathit{Set} \mid \mathit{Condition})$$

is actually syntactic sugar for

$$\mathrm{count}(1 \ldots \mathrm{sup})(x : \mathit{Set} \mid \mathit{Condition})$$

Example 9. The formula

$$\forall(v_1 < v_2 : \mathit{Varieties})\ \mathrm{count}(\lambda)(j : \mathit{Blocks} \mid \mathit{BIBD}(v_1, j) \wedge \mathit{BIBD}(v_2, j))$$

says that each ordered pair of varieties v_1 and v_2 occurs together in exactly λ blocks, via the *BIBD* relation. Regarding the excerpt '$v_1 < v_2 : \mathit{Varieties}$', note that multiple local variables can be quantified at the same time, and that a filtering condition on them may then be pushed across the '$|$' symbol.

Example 10. Assuming that the function *Schedule* is of the domain of Example 3 and thus returns a group, the formula

$$\forall(p_1 < p_2 : \mathit{Players})\ \mathrm{count}(0 \ldots 1)(v : \mathit{Weeks} \mid \mathit{Schedule}(p_1, v) = \mathit{Schedule}(p_2, v))$$

says that there is at most one week where any ordered pair of players p_1 and p_2 is scheduled to play in the same group.

A *cost expression* is a numeric expression that has to be optimised. The *constraints* on the decision variables of a model are a conjunction of formulas, using $\wedge$ as the connective. The *objective* of a model is either to solve its constraints:

$$\text{solve } \mathit{Constraints}$$

or to minimise the value of its cost expression subject to its constraints:

$$\text{minimise } \mathit{CostExpression} \text{ such that } \mathit{Constraints}$$

or similarly for maximising. A *model* consists of a sequence of domain, constant, and decision-variable declarations followed by an objective, without separators.

Example 11. Putting together code fragments from Examples 1, 4, 5, and 9, we obtain the model of Figure 2 two pages ahead, discussed in Section 2.3.

The grammar of ESRA is described in Figure 1. For brevity and ease of reading, we have omitted most syntactic-sugar options as well as the rules for identifiers, names, and numbers. The notation $\langle nt\rangle^{s^*}$ stands for a sequence of zero or more occurrences of the non-terminal $\langle nt\rangle$, separated by symbol s. Similarly, $\langle nt\rangle^{s^+}$ stands for one or more occurrences of $\langle nt\rangle$, separated by s. The typing rules ensure that the equality predicates $=$ and $\neq$ are only applied to expressions of the same type, that the other comparison predicates, such as $\leq$, are only applied to numeric expressions, and so on.

2.3 Examples

We now showcase the elegance and flexibility of our language on three real-life problems, namely Balanced Incomplete Block Designs, the Social Golfers problem, and the Progressive Party problem.

$\langle \mathit{Model}\rangle ::= \langle \mathit{Decl}\rangle^{+}\ \langle \mathit{Objective}\rangle$

$\langle \mathit{Decl}\rangle ::= \langle \mathit{DomDecl}\rangle \mid \langle \mathit{CstDecl}\rangle \mid \langle \mathit{VarDecl}\rangle$

$\langle \mathit{DomDecl}\rangle ::=$ `dom` $\langle \mathit{Id}\rangle$ [`=` $\langle \mathit{Set}\rangle$]

$\langle \mathit{CstDecl}\rangle ::=$ `cst` $\langle \mathit{Id}\rangle$ [`=` $\langle \mathit{Tuple}\rangle \mid \langle \mathit{Set}\rangle$] `:` $\langle \mathit{SetExpr}\rangle$

$\langle \mathit{VarDecl}\rangle ::=$ `var` $\langle \mathit{Id}\rangle$ `:` $\langle \mathit{SetExpr}\rangle$

$\langle \mathit{Objective}\rangle ::=$ `solve` $\langle \mathit{Formula}\rangle$
$\mid$ `(` `minimise` $\mid$ `maximise` `)` $\langle \mathit{NumExpr}\rangle$ `such that` $\langle \mathit{Formula}\rangle$

$\langle \mathit{Expr}\rangle ::= \langle \mathit{Id}\rangle \mid \langle \mathit{Name}\rangle \mid \langle \mathit{Tuple}\rangle \mid \langle \mathit{NumExpr}\rangle \mid \langle \mathit{SetExpr}\rangle \mid \langle \mathit{FuncAppl}\rangle \mid$ `(` $\langle \mathit{Expr}\rangle$ `)`

$\langle \mathit{NumExpr}\rangle ::= \langle \mathit{Id}\rangle \mid \langle \mathit{Int}\rangle \mid \langle \mathit{Nat}\rangle \mid$ `inf` $\mid$ `sup` $\mid \langle \mathit{FuncAppl}\rangle$
$\mid \langle \mathit{NumExpr}\rangle$ `(` `+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%` `)` $\langle \mathit{NumExpr}\rangle$
$\mid$ `(` `-` $\mid$ `abs` `)` $\langle \mathit{NumExpr}\rangle$
$\mid$ `card` $\langle \mathit{SetExpr}\rangle$
$\mid$ `sum` `(` $\langle \mathit{QuantExpr}\rangle$ `)` `(` $\langle \mathit{NumExpr}\rangle$ `)`

$\langle \mathit{SetExpr}\rangle ::= \langle \mathit{Set}\rangle \mid \langle \mathit{SetExpr}\rangle$ `[`$\langle \mathit{Set}\rangle$`]`
$\mid \langle \mathit{SetExpr}\rangle$ `(` `[`[$\langle \mathit{Set}\rangle$]`#`[$\langle \mathit{Set}\rangle$]`]` $\mid$ `#` `)` $\langle \mathit{SetExpr}\rangle$
$\mid \langle \mathit{SetExpr}\rangle$ `(` `[->`[$\langle \mathit{Set}\rangle$]`]` $\mid$ `->` $\mid$ `[+>`[$\langle \mathit{Set}\rangle$]`]` $\mid$ `+>` `)` $\langle \mathit{SetExpr}\rangle$

$\langle \mathit{Set}\rangle ::= \langle \mathit{Id}\rangle \mid$ `int` $\mid$ `nat`
$\mid$ `{` $\langle \mathit{Tuple}\rangle$`,`* `}` $\mid$ `{` $\langle \mathit{ComprExpr}\rangle$ `}`
$\mid \langle \mathit{NumExpr}\rangle$`..`$\langle \mathit{NumExpr}\rangle \mid \langle \mathit{NumExpr}\rangle$

$\langle \mathit{ComprExpr}\rangle ::= \langle \mathit{Expr}\rangle$ `|` `(` $\langle \mathit{IdTuple}\rangle$`&`$^{+}$ `in` $\langle \mathit{SetExpr}\rangle$ `)`$^{\texttt{/\textbackslash}+}$ [`|` $\langle \mathit{Formula}\rangle$]

$\langle \mathit{FuncAppl}\rangle ::= \langle \mathit{Id}\rangle\ \langle \mathit{Tuple}\rangle$

$\langle \mathit{Tuple}\rangle ::=$ `(`$\langle \mathit{Expr}\rangle$`,`$^{+}$`)` $\mid \langle \mathit{Expr}\rangle$

$\langle \mathit{Formula}\rangle ::=$ `true` $\mid$ `false` $\mid \langle \mathit{RelAppl}\rangle$
$\mid \langle \mathit{Formula}\rangle$ `(` `/\` $\mid$ `\/` $\mid$ `=>` $\mid$ `<=` $\mid$ `<=>` `)` $\langle \mathit{Formula}\rangle$
$\mid \langle \mathit{NumExpr}\rangle$ `(` `<` $\mid$ `=<` $\mid$ `=` $\mid$ `>=` $\mid$ `>` $\mid$ `!=` `)` $\langle \mathit{NumExpr}\rangle$
$\mid$ `forall` `(` $\langle \mathit{QuantExpr}\rangle$ `)` `(` $\langle \mathit{Formula}\rangle$ `)`
$\mid$ `count` `(` $\langle \mathit{Set}\rangle$ `)` `(` $\langle \mathit{QuantExpr}\rangle$ `)`

$\langle \mathit{RelAppl}\rangle ::= \langle \mathit{Id}\rangle\ \langle \mathit{Tuple}\rangle$

$\langle \mathit{QuantExpr}\rangle ::=$ `(` `(` $\langle \mathit{RelQvars}\rangle \mid \langle \mathit{IdTuple}\rangle$`&`$^{+}$ `)` `in` $\langle \mathit{SetExpr}\rangle$ `)``,`$^{+}$ [ `|` $\langle \mathit{Formula}\rangle$]

$\langle \mathit{RelQvars}\rangle ::= \langle \mathit{Expr}\rangle$ `(` `<` $\mid$ `=<` $\mid$ `=` $\mid$ `>=` $\mid$ `>` $\mid$ `!=` `)` $\langle \mathit{Expr}\rangle$

$\langle \mathit{IdTuple}\rangle ::= \langle \mathit{Id}\rangle \mid$ `(` $\langle \mathit{Id}\rangle$`,`$^{+}$ `)`

Fig. 1. The grammar of ESRA

dom *Varieties*, *Blocks*
cst $r, k, \lambda : \mathbb{N}$
var *BIBD* : *Varieties* ${}^{r}\!\times^{k}$ *Blocks*
solve
    $\forall(v_1 < v_2 : \mathit{Varieties})\ \mathrm{count}(\lambda)(j : \mathit{Blocks} \mid \mathit{BIBD}(v_1, j) \wedge \mathit{BIBD}(v_2, j))$

Fig. 2. A pretty-printed ESRA model for BIBDs

```
dom Varieties, Blocks
cst r, k, lambda : nat
var BIBD : Varieties [r#k] Blocks
solve
   forall (v1 < v2 : Varieties)
      count (lambda) (j : Blocks | BIBD(v1,j) /\ BIBD(v2,j))
```

Fig. 3. An ESRA model for BIBDs

Balanced Incomplete Block Designs. Let V be any set of v elements, called *varieties*. A *balanced incomplete block design* (BIBD) is a bag of b subsets of V, called *blocks*, each of size k (constraint C_1), such that each pair of distinct varieties occurs together in exactly λ blocks (C_2), with $2 \leq k < v$. An implied constraint is that each variety occurs in the same number of blocks (C_3), namely $r = \lambda(v-1)/(k-1)$. A BIBD is parameterised by a 5-tuple $\langle v, b, r, k, \lambda \rangle$ of parameters. Originally intended for the design of statistical experiments, BIBDs also have applications in cryptography and other domains. See Problem 28 at `http://www.csplib.org` for more information.

The instance data can be declared as the two domains *Varieties* and *Blocks*, of implicit sizes v and b respectively, as well as the three natural-number constants r, k, and λ, as in Examples 1 and 4. A unique relational decision variable, *BIBD*, can then be declared as in Example 5, thereby immediately taking care of the constraints C_1 and C_3. The remaining constraint C_2 can be modelled as in Example 9. Figure 2 shows the resulting pretty-printed ESRA model, while Figure 3 shows it in ASCII notation.

For comparison, an OPL [31] model is shown in Figure 4, where '`= ...`' means that the value is to be found in a corresponding datafile. The decision variable `BIBD` is a 2-dimensional array of integers `0` or `1`, indexed by the varieties and blocks, such that `BIBD[i,j]` = `1` iff variety `i` is contained in block `j`. Furthermore, the constraints C_1 and C_3, which we could capture by multiplicities in the ESRA model, need here to be stated in more length. Finally, the constraint C_2 is stated using a higher-order constraint:[2] for each ordered pair of varieties `v1` and `v2`, the number of times they appear in the same block, that is the number of blocks `j` where `BIBD(v1,j)` = `1` = `BIBD(v2,j)` holds, must equal `lambda`.

In an OPL model, one needs to decide what concrete datatypes to use for representing the abstract decision variables of the original problem statement.

[2] A higher-order constraint refers to the truth value of another constraint. In OPL, the latter is nested in parentheses, truth is represented by 1, and falsity by 0.

```
enum Varieties = ..., Blocks = ...;
int r = ...; int k = ...; int lambda = ...;
range Boolean 0..1;
var Boolean BIBD[Varieties,Blocks];
solve {
   forall(j in Blocks) sum(i in Varieties) BIBD[i,j] = k;
   forall(i in Varieties) sum(j in Blocks) BIBD[i,j] = r;
   forall(ordered v1,v2 in Varieties)
      sum(j in Blocks) (BIBD[v1,j] = 1 = BIBD[v2,j]) = lambda;
   ... symmetry-breaking code ...
};
```

Fig. 4. An OPL model for BIBDs

In this case, we chose a 2-dimensional `0/1` array `BIBD`, indexed by `Varieties` and `Blocks`. We could just as well have chosen a different representation, say (if OPL had set variables) a 1-dimensional array `BIBD`, indexed by `Blocks`, of subsets of `Varieties`. Such a choice affects the formulation of every constraint and the cost expression, but is premature as even expert intuition is weak in predicting which representation choice leads to the best solving efficiency. Consequently, the modeller has to frequently reformulate the constraints and the cost expression while experimenting with different representations. No such choices have to be made in an ESRA model, making ESRA a more convenient modelling language.

As a consequence to such representation choices, one often introduces an astronomical amount of symmetries into an OPL model that are not present in the original problem statement [10]. For example, given a solution, any two rows or columns in the array `BIBD` can be swapped, giving a different, but symmetrically equivalent, solution. Such symmetries need to be addressed in order to achieve efficient solving. Hence, symmetry-breaking code [10,32] would have to be inserted, as indicated in Figure 4. Since such choices are postponed to the compilation phase in ESRA (see Section 2.4), any symmetries consciously introduced can be handled (automatically) in that process.

The Social Golfers Problem. In a golf club, there are n players, each of whom plays golf once a week (constraint C_1) and always in g groups of size s (C_2), hence $n = gs$. The objective is to determine whether there is a schedule of w weeks of play for these golfers, such that there is at most one week where any two distinct players are scheduled to play in the same group (C_3). An implied constraint is that every group occurs exactly sw times across the schedule (C_4). See Problem 10 at `http://www.csplib.org` for more information.

The instance data can be declared as the three natural-number constants g, s, and w, via 'cst $g, s, w : \mathbb{N}$', as well as the three domains *Players*, *Weeks*, and *Blocks*, as in Example 1. A unique decision variable, *Schedule*, can then be declared using the functional domain in Example 3, thereby immediately taking care of the constraints C_1 (because of the totality of the function) and C_4. The

cst $g, s, w : \mathbb{N}$
dom $\mathit{Players} = 1 \ldots g * s,\ \mathit{Weeks} = 1 \ldots w,\ \mathit{Groups} = 1 \ldots g$
var $\mathit{Schedule} : (\mathit{Players} \times \mathit{Weeks}) \longrightarrow^{s*w} \mathit{Groups}$
solve
$\quad \forall (p_1 < p_2 : \mathit{Players})\ \mathrm{count}(0 \ldots 1)(v : \mathit{Weeks} \mid \mathit{Schedule}(p_1, v) = \mathit{Schedule}(p_2, v))$
$\quad \wedge\ \forall (h : \mathit{Groups}, v : \mathit{Weeks})\ \mathrm{count}(s)(p : \mathit{Players} \mid \mathit{Schedule}(p, v) = h)$

Fig. 5. A pretty-printed ESRA model for the Social Golfers problem

constraint C_3 can be modelled as in Example 10. The constraint C_2 can be stated using the count quantifier, as seen in the pretty-printed ESRA model of Figure 5.

Note the different style of modelling sets of unnamed objects, via the separation of models from the instance data, compared to Figure 2. There we introduce two sets without initialising them at the model level, while here we introduce three uninitialised constants that are then used to arbitrarily initialise three domains of desired cardinalities. Both models can be reformulated in the other style. The benefit of such sets of unnamed objects is that their elements are indistinguishable, so that lower-level representations of relational decision variables whose domains involve such sets are known to introduce symmetries.

The Progressive Party Problem. The problem is to timetable a party at a yacht club. Certain boats are designated as hosts, while the crews of the remaining boats are designated as guests. The crew of a host boat remains on board throughout the party to act as hosts, while the crew of a guest boat together visits host boats over a number of periods. The spare capacity of any host boat is not to be exceeded at any period by the sum of the crew sizes of all the guest boats that are scheduled to visit it then (constraint C_1). Any guest crew can visit any host boat in at most one period (C_2). Any two distinct guest crews can visit the same host boat in at most one period (C_3). See Problem 13 at `http://www.csplib.org` for more information.

The instance data can be declared as the three domains *Guests*, *Hosts*, and *Periods*, via 'dom *Guests*, *Hosts*, *Periods*', as well as the two functional constants *SpareCap* and *CrewSize*, as in Example 4. A unique functional decision variable, *Schedule*, can then be declared via 'var $\mathit{Schedule} : (\mathit{Guests} \times \mathit{Periods}) \longrightarrow \mathit{Hosts}$'. The constraint C_1 can now be modelled as in Example 8. The constraints C_2 and C_3 can be stated using the count quantifier, as seen in the pretty-printed ESRA model of Figure 6.

2.4 Compiling Relational Models

A compiler for ESRA is currently under development. It is being written in OCAML (`http://www.ocaml.org`) and compiles ESRA models into SICStus Prolog [5] finite-domain constraint programs. Our choice of target language is motivated by its excellent collection of global constraints and by our collaboration with its developers on designing new global constraints.

dom *Guests*, *Hosts*, *Periods*
cst *SpareCap* : *Hosts* $\longrightarrow \mathbb{N}$, *CrewSize* : *Guests* $\longrightarrow \mathbb{N}$
var *Schedule* : $(\mathit{Guests} \times \mathit{Periods}) \longrightarrow \mathit{Hosts}$
solve

$$\forall(p : \mathit{Periods},\ h : \mathit{Hosts}) \left(\sum_{g:\mathit{Guests}\ \mid\ \mathit{Schedule}(g,p)=h} \mathit{CrewSize}(g) \right) \leq \mathit{SpareCap}(h)$$
$$\wedge\ \forall(g : \mathit{Guests}, h : \mathit{Hosts})\ \mathrm{count}(0\ldots1)(p : \mathit{Periods} \mid \mathit{Schedule}(g,p) = h)$$
$$\wedge\ \forall(g_1 < g_2 : \mathit{Guests})\ \mathrm{count}(0\ldots1)(p : \mathit{Periods} \mid \mathit{Schedule}(g_1,p) = \mathit{Schedule}(g_2,p))$$

Fig. 6. A pretty-printed ESRA model for the Progressive Party problem

We already have an ESRA-to-OPL compiler [36, 15], written in Java, for a restriction of ESRA to functions, now called Functional-ESRA. That project gave us much of the expertise needed for developing the current compiler.

The solver-independent ESRA language is so high-level that it is very small compared to such target languages, especially in the number of necessary primitive constraints. The full panoply of features of such target languages can, and must, be deployed during compilation. In particular, the implementation of decision-variable indices into matrices is well-understood.

In order to bootstrap our new compiler quickly, we decided to represent initially *every* relational decision variable by a matrix of 0/1 variables, indexed by its participating sets. This first version of the new compiler is thus deterministic.

The plan is then to add alternatives to this unique representation rule, depending on the multiplicities and other constraints on the relation, achieving a *non-deterministic compiler*, such as our existing Functional-ESRA-to-OPL compiler [36, 15]. The modeller is then invited to experiment with her (real-life) instance data and the resulting compiled programs, so as to determine which one is the 'best'. If the compiler is provided with those instance data, then it can be extended to automate such experiments and generate rankings.

Eventually, more intelligence will be built into the compiler via *heuristics* (such as those of [15]) for the compiler to rank the resulting compiled programs by decreasing likelihood of efficiency, without any recourse to experiments. Indeed, depending on the multiplicities and other constraints on a relation, certain representations thereof can be shown to be better than others, under certain assumptions on the targeted solver, and this either theoretically (see for instance [33] for bijections and [15] for injections) or empirically (see for instance [28] for bijections). We envisage a hybrid interactive/heuristic compiler.

Our ultimate aim is of course to design an actual *solver for relational constraints*, without going through compilation.

3 Benefits of Relational Modelling

In our experience, and as demonstrated in Section 2.3, a relational constraint modelling language leads to more *concise and intuitive models*, as well as to more *efficient and effective model formulation and verification*. Due to ESRA being *smaller* than conventional constraint programming languages, we believe it is easier to learn and master, making it a good candidate for a teaching medium. All this could entail a better dissemination of constraint technology.

Relational languages seem a good trade-off between generality and specificity, enabling *efficient solving* despite more generality. Relations are a *single*, powerful concept for elegantly modelling many aspects of combinatorial problems. Also, there are *not too many* different, and even *standard*, ways of representing relations and relational expressions. Known and future modelling insights, such as those in [15, 28, 33], can be built into the compilers, so that even time-pressed or less competent modellers can benefit from them. Modelling is unencumbered by early if not uninformed commitments to representation choices. Low-level modelling devices such as reification and higher-order constraints can be encapsulated as implementation devices. The number of decision variables being reduced, there is even hope that directly solving the constraints at the high relational level can be faster than solving their compiled lower-level counterparts. All this illustrates that more generality need not mean poorer performance.

Relational models are more amenable to *maintenance* when the combinatorial problem changes, because most of the tedium is taken care of by the compiler. Model maintenance at the relational level reduces to adapting to the new problem, with all representation (and solving) issues left to the compiler. Very little work is involved here when a multiplicity change entails a preferable representation change for a relation. Maintenance can even be necessary when the statistical distribution of the problem instances that are to be solved changes [22]. If information on the new distribution is given to the envisaged compiler, a simple recompilation will take care of the maintenance.

Relational models are at a more suitable level for possibly automated model *reformulation*, such as via the inference and selection of suitable *implied constraints*, with again the compiler assisting in the more mundane aspects. In the BIBD and Social Golfers examples, we have observed that multiplicities provide a nice framework for discovering and stating some implied constraints. Indeed, the language makes the modeller think about making these multiplicities explicit, even if they were not in the original problem formulation.

Relational models are more amenable to *constraint analysis*. Detected properties as well as properties consciously introduced during compilation into lower-level programs, such as symmetry or bijectiveness, can then be taken into account during compilation [10], especially using tractability results [32].

There would be further benefits to an abstract modelling language if it were adopted as a *standard front-end language* for solvers. Models and instance data would then be *solver-independent* and could be shared between solvers, whatever their technology. Indeed, the targeted solvers need not even use constraint technology, but could just as well use answer-set programming, linear programming, local search, or propositional satisfiability technology, or any hybrid thereof. This would facilitate fair and homogeneous comparisons, say via new standard benchmarks, as well as foster competition in fine-tuning the compilers.

4 Conclusion

We have argued that solver-independent, abstract constraint modelling leads to a simpler and smaller language; to more concise, intuitive, and analysable models; as well as to more efficient and effective model formulation, maintenance, reformulation, and verification. All this can be achieved without sacrificing the possibility of efficient solving, so that even time-pressed or less competent modellers can be well assisted. Towards this, we have proposed the ESRA relational modelling language, showcased its elegance on some well-known problems, and outlined a compilation philosophy for such languages. To conclude, let us look at related work (Section 4.1) and future work(Section 4.2).

4.1 Related Work

We have here generalised and re-engineered our own work [11, 36, 15] on a predecessor of ESRA, now called Functional-ESRA, that only supports functional decision variables, by pursuing the aim of relational modelling outlined in [9]. Elsewhere, such ideas have recently inspired a related project [3], incorporating partition decision variables. Constraints for bag decision variables [2, 7, 34] and sequence decision variables [2, 26] have also been proposed.

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 [16], which itself gained much from the Z specification notation [29] (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, the commercial OPL [31] stands out as a medium-level modelling language and actually gave the impetus to design ESRA: see the BIBD example in Section 2.3 and consult [9] for a further comparison of elegant ESRA models with more awkward (published) OPL counterparts that do not provide all the benefits of Section 3. Other higher-level constraint modelling languages than ESRA have been proposed, such as ALICE [18], $CLP(Fun(D))$ [14], CLPS [2], CONJUNTO [13], EACL [30], $\{log\}$ [7], NCL [37], and the language of [24]. Our ESRA shares with them the quest for a practical declarative modelling language based on a strongly-typed fuller first-order logic than Horn clauses, with sequence, set, bag, functional, or even relational decision variables, while often dispensing with recursion, negation, and unbounded quantification. However, ESRA goes way beyond them, by advocating an ADT view (of relations), so that representations need not be fixed in advance, by providing an elegant notation for multiplicity constraints, and by promising intelligent compilation.

In the field of knowledge representation, answer-set programming (ASP) has recently been advocated [21] as a practical constraint solving paradigm, especially for dynamic domains such as planning. A set of (disjunctive) function-free clauses, where classical negation and negation as failure are allowed, is interpreted as a constraint, stating when an atom is in a solution, called an answer set or a stable model. This non-monotonic approach differs from constraint (logic) programming, where statements are used to add atomic constraints on decision

dom *Cities*
cst *Distance* : $(\mathit{Cities} \times \mathit{Cities}) \longrightarrow \mathbb{N}$
var *Next* : $\mathit{Cities} \longrightarrow^1 \mathit{Cities}$
minimise $\sum_{c:\mathit{Cities}} \mathit{Distance}(c, \mathit{Next}(c))$
such that $\forall (c_1 \& c_2 : \mathit{Cities})\ \mathit{Next}^*(c_1) = c_2$

Fig. 7. A pretty-printed ESRA model for the Travelling Salesperson problem

variables to a constraint store, whereupon propagation and search are used to construct solutions. Implementation methods for computing the answer sets of ground programs have advanced significantly over recent years, possibly using propositional satisfiability (SAT) solvers. Also, effective grounding procedures have been devised for some classes of such programs with (schematic) variables. Sample ASP systems are DLV [19] and SMODELS [23]. Closely related are *ConstraintLingo* [8] and NP-SPEC [4]. The languages of these systems include useful features, such as cardinality and weight constraints, aggregate functions, and soft constraints. They have strictly more expressive power than propositional logic and traditional constraint (logic) programming/modelling languages, including ESRA. Again, our objective only is a language that is useful for elegantly modelling a large number of combinatorial problems. The cardinality constraint of SMODELS is a restriction of the ESRA 'count' quantifier to interval multiplicities, as opposed to set multiplicities. Speed comparisons with SAT solvers were encouraging, but no comparison has been done yet with constraint solvers.

4.2 Future Work

Most of our future work has already been listed in Sections 2.4 and 3 about the compiler design and long-term benefits of relational modelling, such as the generation of implied constraints and the breaking of symmetries.

We have argued that our ESRA language is very small. This is mostly because we have not yet identified the need for any other operators or predicates. An exception to this is the need for *transitive closure relation constructors.* We aim at modelling the well-known Travelling Salesperson (TSP) problem as in Figure 7, where the transitive closure of the bijection *Next* on *Cities* is denoted by *Next**. This general mechanism avoids the introduction of an *ad hoc* 'circuit' constraint as in ALICE [18].

As we do not aim at a complete constraint modelling language, we can be very conservative in what missing features shall be added to ESRA when they are identified. Also, for manpower reasons, we do not yet propose other ADTs, say for bags or sequences, although this was originally part of our original vision (see Section 3.3 of [11]).

Our request for explicit model-level distinction between constants and decision variables may be eventually lifted, as the default is run-time initialisation: we could treat as constants any universally quantified variable that was actually initialised and treat all the others as decision variables. This requires a convincing example, though, as well as just-in-time compilation.

In [20], a type system is derived for binary relations that can be used as an input to specialised filtering algorithms. This kind of analysis can be integrated into the *relational solver* we have in mind.

Also, a *graphical language* could be developed for the data modelling, including the multiplicity constraints on relations, so that only the cost expression and the constraints would need to be textually expressed.

Finally, a *search language*, such as SALSA [17] or the one of OPL [31], but at the level of relational modelling, should be adjoined to the constraint modelling language proposed here, so that more expert modellers can express their own search heuristics.

Acknowledgements. This work is partially supported by grant 221-99-369 of VR, the Swedish Research Council, and by institutional grant IG2001-67 of STINT, the Swedish Foundation for International Cooperation in Research and Higher Education. We thank Nicolas Beldiceanu, Mats Carlsson, Esra Erdem, Brahim Hnich, Daniel Jackson, Zeynep Kızıltan, François Laburthe, Gerrit Renker, Christian Schulte, Mark Wallace, and Simon Wrang for stimulating discussions, as well as the constructive reviewers of previous versions of this paper.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. F. Ambert, B. Legeard, and E. Legros. Programmation en logique avec contraintes sur ensembles et multi-ensembles héréditairement finis. *Techniques et Sciences Informatiques*, 15(3):297–328, 1996.
3. A. Bakewell, A. M. Frisch, and I. Miguel. Towards automatic modelling of constraint satisfaction problems: A system based on compositional refinement. In *Proceedings of the 2nd International Workshop on Modelling and Reformulating CSPs*, pages 3–17, 2003. Available at `http://www-users.cs.york.ac.uk/~frisch/Reformulation/03/`.
4. M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NPSPEC: An executable specification language for solving all problems in NP. In G. Gupta, editor, *Proceedings of PADL'99*, volume 1551 of *LNCS*, pages 16–30. Springer-Verlag, 1999.
5. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of PLILP'97*, number 1292 in LNCS, pages 191–206. Springer-Verlag, 1997.
6. M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In *Proceedings of ICLP'01*, volume 2237 of *LNCS*, pages 212–226. Springer-Verlag, 2001.
7. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931, 2000.

8. R. Finkel, V. Marek, and M. Truszczyński. Tabular constraint-satisfaction problems and answer-set programming. In *Proceedings of the AAAI Spring Symposium on Answer-Set Programming*, 2001. Available at `http://www.cs.nmsu.edu/~tson/ASP2001/`.
9. P. Flener. Towards relational modelling of combinatorial optimisation problems. In C. Bessière, editor, *Proceedings of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, pages 31–38, 2001. Available at `http://www.lirmm.fr/~bessiere/ws_ijcai01/`.
10. P. Flener, A. M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, and T. Walsh. Matrix modelling: Exploiting common patterns in constraint programming. In *Proc. of the 1st Int'l Workshop on Reformulating CSPs*, pages 27–41, 2002. Available at `http://www-users.cs.york.ac.uk/~frisch/Reformulation/02/`.
11. P. Flener, B. Hnich, and Z. Kızıltan. Compiling high-level type constructors in constraint programming. In I. Ramakrishnan, editor, *Proceedings of PADL'01*, volume 1990 of *LNCS*, pages 229–244. Springer-Verlag, 2001.
12. P. Flener, J. Pearson, and M. Ågren. The Syntax, Semantics, and Type System of esra. Technical report, ASTRA group, April 2003. Available at `http://www.it.uu.se/research/group/astra/`.
13. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
14. T. J. Hickey. Functional constraints in CLP languages. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 355–381. The MIT Press, 1993.
15. B. Hnich. *Function Variables for Constraint Programming.* PhD thesis, Department of Information Science, Uppsala University, Sweden, 2003. Available at `http://publications.uu.se/theses/`.
16. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. *Software Engineering Notes*, 26(5):62–73, 2001. Proceedings of FSE/ESEC'01.
17. F. Laburthe and Y. Caseau. SALSA: A language for search algorithms. *Constraints*, 7:255–288, 2002.
18. J.-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.
19. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. In *ACM Transactions on Computational Logic*, forthcoming. Available at `http://arxiv.org/ps/cs.AI/0211004`.
20. D. Lesaint. Inferring constraint types in constraint programming. In P. Van Hentenryck, editor, *Proceedings of CP'02*, volume 2470 of *LNCS*, pages 492–507. Springer-Verlag, 2002.
21. V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.
22. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1–2):7–43, 1996.
23. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and AI*, 25(3–4):241–273, 1999.
24. N. Pelov and M. Bruynooghe. Extending constraint logic programming with open functions. In *Proceedings of PPDP'00*, pages 235–244. ACM Press, 2000.
25. N. Pelov, M. Denecker, and M. Bruynooghe. Partial stable models for logic programs with aggregates. In *Proceedings of LPNMR'04*, volume 2923 of *LNCS*, pages 207–219. Springer-Verlag, 2004.

26. G. Pesant. A regular language membership constraint for sequences of variables. In *Proceedings of the 2nd International Workshop on Modelling and Reformulating CSPs*, pages 110–119, 2003. Available at `http://www-users.cs.york.ac.uk/~frisch/Reformulation/03/`.
27. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
28. B. M. Smith. Modelling a permutation problem. Technical Report 18, School of Computing, University of Leeds, UK, 2000. Also in *Proceedings of the ECAI'00 Workshop on Modelling and Solving Problems with Constraints*.
29. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
30. E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer-aided constraint programming system. In J. Little, editor, *Proceedings of PACLP'99*, pages 81–93. The Practical Application Company, 1999.
31. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
32. P. Van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Tractable symmetry breaking for CSPs with interchangeable values. In *Proceedings of IJCAI'03*, pages 277–282. Morgan Kaufmann, 2003.
33. T. Walsh. Permutation problems and channelling constraints. In R. Nieuwenhuis and A. Voronkov, editors, *Proc. of LPAR'01*, number 2250 in LNCS, pages 377–391. Springer-Verlag, 2001.
34. T. Walsh. Consistency and propagation with multiset constraints: A formal viewpoint. In F. Rossi, editor, *Proceedings of CP'03*, number 2833 in LNCS, pages 724–738. Springer-Verlag, 2003.
35. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
36. S. Wrang. Implementation of the ESRA Constraint Modelling Language. Master's thesis, Master's Thesis in Computing Science 223, Department of Information Technology, Uppsala University, Sweden, 2002. Available at `ftp://ftp.csd.uu.se/pub/papers/masters-theses/`.
37. J. Zhou. Introduction to the constraint language NCL. *Journal of Logic Programming*, 45(1–3):71–103, 2000.