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, and all this without sacrificing the possibility of efficient solving, so that even lazy 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 higher-level, simpler, and smaller modelling notation is needed.

In our opinion, even recent commercial languages such as OPL [20] 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.

There is much to be learned from formal methods and semantic modelling. 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 [18] specification languages, the ALLOY [11] object modelling language, and the *Object Constraint Language* (OCL) [23] of the *Unified Modelling Language* (UML) [16]. In semantic data modelling this had been long advocated; most notably via entity-relationship-attribute (ERA) diagrams.

* The authors' names are ordered according to the Swedish alphabet.
** An extended abstract of this paper appears in the pre-proceedings of LOPSTR'03.

Sets and set expressions recently started appearing as modelling devices in some constraint programming languages. Set variables are often implemented by the set interval representation [8]. In the absence of such an explicit set concept, modellers usually painstakingly represent a set variable as a sequence of 0/1 integer variables, as long as the domain of the set.

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 in many ways, say 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. Other than retrieving the (unique) image under a total function of a domain element, there has been no support for relational expressions.

Matrix modelling [5] has been advocated as one way of capturing common modelling patterns. Alternatively, it has been argued [6, 10] 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 claim, as in [4], that a suitable first-order relational calculus is a good basis for a high-level ADT-based 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 high-level 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 [7] and a prototype of the advocated compiler is currently under implementation.

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 Modelling with ESRA

In Section 2.1, we justify the design decisions behind our new ESRA modelling language. 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 compiler 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 must be 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 no negation and no 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 [7] for a complete description of the language. Code examples are 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 function* 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 '...' (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 $0 \ldots \mathrm{sup}$ and $\mathrm{inf} \ldots \mathrm{sup}$ 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 inline, using the '=' symbol, or at run-time, via a datafile, otherwise interactively.

Example 1. The declaration

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

declares two domains called *Varieties* and *Blocks* that are to be initialised at run-time. Similarly, the declaration

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

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 in-line. Finally, the declaration

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

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

The usual binary infix $\times$ domain constructor (denoted in ASCII by '`#`') allows the construction of Cartesian products, so that relations can be declared of this *constructed domain*. Consider the relation domain $A \times B$; then A and B must be domains, designating the participant sets of any relation in $A \times B$.

In order to capture frequently occurring multiplicity constraints on relations, we offer a parameterised binary infix $\times$ domain constructor. Consider the relation domain $A \ {}^{M_1}\!\times^{M_2} B$. The conditions on A and B are as above. 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 domain '*Varieties* $^{r}\!\times^{k}$ *Blocks*' designates the set of all relations in *Varieties* $\times$ *Blocks* where every variety occurs in exactly r blocks and every block contains exactly k varieties. These are also two examples where an integer abbreviates the singleton domain containing it.

In the absence of such facilities for relations and their multiplicities, a relation domain would have to be declared using arrays, say. This may constitute 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 relation decision variable will lead to the most efficient solving. The problem constraints, including the multiplicities, would have to be enforced further down in the model, based on the chosen representation. If the experiments revealed that another representation should be tried, the modeller would have to first painstakingly rewrite the declaration of the decision variable as well as all the constraints on it. 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 modelling insights. Furthermore, multiplicities need not become counting constraints, but are succinctly and conveniently captured in the declaration.

We view sets as unary relations. So $A \ M$, where A is a domain and M an integer set, constructs the domain of all subsets of A whose cardinality is in M.

For total and partial functions the left-hand multiplicity M_1 is $1 \ldots 1$ and $0 \ldots 1$ 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

[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.

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 $0 \dots 1$, $1 \dots \sup$, 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 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. Note the nesting in this domain: the Cartesian product $\mathit{Players} \times \mathit{Weeks}$ is the left-hand argument of the outer Cartesian product.

We provide no support for multisets and sequences. Note that a *multiset* can be modelled as a total function from its domain into $\mathbb{N}$, giving the multiplicity 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 multisets and sequences is fixed (to the one of total functions), because, as we shall see in Section 2.4, the relations (and thus total functions) of a model need not have the same representation.

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

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 '$\in$' (denoted in ASCII by '`in`'), meaning set membership, or by ':' (which is often used in mathematics and logic for '$\subseteq$'), meaning set inclusion. Constants are initialised in-line, using the '=' symbol, or at run-time, via a datafile, otherwise interactively. Run-time initialisation provides a neat separation of problem models and problem instances.

Example 4. The declaration 'cst $r, k, \lambda \in \mathbb{N}$' declares three natural number constants that are to be initialised at run-time. As already seen in Example 2, the availability of total functions makes arrays unnecessary. The declaration

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

declares that the given crew sizes of the guests as well as the given spare capacities of the hosts are natural numbers, to be provided at run-time.

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

Example 5. The declaration 'var $\mathit{BIBD} : \mathit{Varieties}\ {}^{r}\!\times^{k}\ \mathit{Blocks}$' declares a relation called *BIBD* of the domain of Example 2. Replacing the : by $\in$ would rather declare a particular *pair* of that domain.

Modelling the Cost Function 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 (intensionally or extensionally) given sets or set identifiers, including the predefined sets $\mathbb{N}$ and $\mathbb{Z}$. Only the 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 (but see the discussion of future work in Section 4).

Example 6. The numeric expression

$$\sum_{g \in Guests \mid Schedule(g,p)=h} CrewSize(g)$$

denotes the sum of the crew sizes of all the guests 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 $CrewSize(g)$ 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 $BIBD(v_1, i)$ 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, models can be formulated positively, making the negation connective unnecessary. The usual typing and precedence rules for operators and connectives apply. All binary operators associate to the left.

Example 8. The formula

$$\forall(p \in Periods,\ h \in Hosts) \left(\sum_{g \in Guests \mid Schedule(g,p)=h} CrewSize(g) \right) \leq SpareCapacity(h)$$

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 boat h at period p.

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

$$\text{count}(\mathit{Multiplicity})(x \in \mathit{Set} \mid \mathit{Condition})$$

to hold if and only if the cardinality of the set comprehension $\{x \in \mathit{Set} \mid \mathit{Condition}\}$ is in the integer set $\mathit{Multiplicity}$. So '$\exists(x \in \mathit{Set} \mid \mathit{Condition})$' is actually syntactic sugar for '$\text{count}(1 \ldots \text{sup})(x \in \mathit{Set} \mid \mathit{Condition})$'.

Example 9. The formula

$$\forall(v_1 < v_2 \in \mathit{Varieties})\ \text{count}(\lambda)(i \in \mathit{Blocks} \mid \mathit{BIBD}(v_1, i) \wedge \mathit{BIBD}(v_2, i))$$

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

Example 10. Recalling from Ex. 3 that *Schedule* returns groups, the formula

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

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

A *cost function* 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 function subject to its constraints:

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

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 type 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. Only one feature of the language has not been described yet, namely projections. We prefer doing so in the semantic context of the Progressive Party problem in Section 2.3.

⟨*Model*⟩ ::= ⟨*Decl*⟩$^{+}$ ⟨*Objective*⟩

⟨*Decl*⟩ ::= ⟨*DomDecl*⟩ | ⟨*CstDecl*⟩ | ⟨*VarDecl*⟩

⟨*DomDecl*⟩ ::= `dom` ⟨*Id*⟩ [`=` ⟨*Set*⟩]

⟨*CstDecl*⟩ ::= `cst` ⟨*Id*⟩ [`=` ⟨*Tuple*⟩ | ⟨*Set*⟩] (`in` | `:`) ⟨*SetExpr*⟩

⟨*VarDecl*⟩ ::= `var` ⟨*Id*⟩ (`in` | `:`) ⟨*SetExpr*⟩ ⟨*ProjClause*⟩$^{/\backslash^{*}}$

⟨*ProjClause*⟩ ::= `where` ⟨*Id*⟩`(` (⟨*Set*⟩ | `_`)`,`$^{+}$ `)` `:` ⟨*SetExpr*⟩

⟨*Objective*⟩ ::= `solve` ⟨*Formula*⟩
| (`minimise` | `maximise`) ⟨*NumExpr*⟩ `such that` ⟨*Formula*⟩

⟨*Expr*⟩ ::= ⟨*Id*⟩ | ⟨*Name*⟩ | ⟨*Tuple*⟩ | ⟨*NumExpr*⟩ | ⟨*SetExpr*⟩ | ⟨*FuncAppl*⟩ | `(` ⟨*Expr*⟩ `)`

⟨*NumExpr*⟩ ::= ⟨*Id*⟩ | ⟨*Int*⟩ | ⟨*Nat*⟩ | `inf` | `sup` | ⟨*FuncAppl*⟩
| ⟨*NumExpr*⟩ (`+` | `-` | `*` | `/` | `%`) ⟨*NumExpr*⟩
| (`-` | `abs`) ⟨*NumExpr*⟩
| `card` ⟨*SetExpr*⟩
| `sum` `(` ⟨*QuantExpr*⟩ `)` `(` ⟨*NumExpr*⟩ `)`

⟨*SetExpr*⟩ ::= ⟨*Set*⟩ | ⟨*SetExpr*⟩ `[`⟨*Set*⟩`]`
| ⟨*SetExpr*⟩ (`[`[⟨*Set*⟩]`#`[⟨*Set*⟩]`]` | `#`) ⟨*SetExpr*⟩
| ⟨*SetExpr*⟩ (`[->`[⟨*Set*⟩]`]` | `->` | `[+>`[⟨*Set*⟩]`]` | `+>`) ⟨*SetExpr*⟩

⟨*Set*⟩ ::= ⟨*Id*⟩ | `int` | `nat`
| `{` ⟨*Tuple*⟩`,`* `}` | `{` ⟨*ComprExpr*⟩ `}`
| ⟨*NumExpr*⟩`..`⟨*NumExpr*⟩ | ⟨*NumExpr*⟩

⟨*ComprExpr*⟩ ::= ⟨*Expr*⟩ | (⟨*IdTuple*⟩$^{\&^{+}}$ `in` ⟨*SetExpr*⟩)$^{/\backslash^{+}}$ [| ⟨*Formula*⟩]

⟨*FuncAppl*⟩ ::= ⟨*Id*⟩ ⟨*Tuple*⟩

⟨*Tuple*⟩ ::= (⟨*Expr*⟩`,`$^{+}$ `)` | ⟨*Expr*⟩

⟨*Formula*⟩ ::= `true` | `false` | ⟨*RelAppl*⟩
| ⟨*Formula*⟩ (`/\` | `\/` | `=>` | `<=` | `<=>`) ⟨*Formula*⟩
| ⟨*Expr*⟩ (`<` | `=<` | `=` | `>=` | `>` | `!=`) ⟨*Expr*⟩
| `forall` `(` ⟨*QuantExpr*⟩ `)` `(` ⟨*Formula*⟩ `)`
| `count` `(` ⟨*Set*⟩ `)` `(` ⟨*QuantExpr*⟩ `)`

⟨*RelAppl*⟩ ::= ⟨*Id*⟩ ⟨*Tuple*⟩

⟨*QuantExpr*⟩ ::= ((⟨*RelQvars*⟩ | ⟨*IdTuple*⟩$^{\&^{+}}$) `in` ⟨*SetExpr*⟩)`,`$^{+}$ [| ⟨*Formula*⟩]

⟨*RelQvars*⟩ ::= ⟨*Expr*⟩ (`<` | `=<` | `=` | `>=` | `>` | `!=`) ⟨*Expr*⟩

⟨*IdTuple*⟩ ::= ⟨*Id*⟩ | `(` ⟨*Id*⟩`,`$^{+}$ `)`

Fig. 1. The grammar of ESRA

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

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

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

Fig. 3. An ESRA model for BIBDs

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.

Balanced Incomplete Block Designs. Let V be any set of v elements, called *varieties*. A *balanced incomplete block design* (BIBD) is a multiset 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 $2 \leq k < v$. Implied constraints are that each variety occurs in the same number of blocks (C_3), namely $r = \lambda(v-1)/(k-1)$, as well as that $bk = vr$ and $\lambda < r$. A BIBD is thus parameterised by a 5-tuple $\langle v, b, r, k, \lambda \rangle$ of parameters, not all of which are independent. Originally intended for the design of statistical experiments, BIBDs also have applications in cryptography and other domains. See Problem 28 at 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 decision variable, *BIBD*, can then be declared using the relational domain 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.

The Social Golfers Problem. In a golf club, there are n players, each of whom play 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 www.csplib.org for more information.

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

Fig. 4. A pretty-printed ESRA model for the Social Golfers problem

The instance data can be declared as the three natural-number constants g, s, and w, via 'cst $g, s, w \in \mathbb{N}$', 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 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 4.

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 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 constant functions *SpareCapacity* and *CrewSize*, as in Example 4. A unique decision variable, *Schedule*, can then be declared via 'var $Schedule : (Guests \times Periods) \longrightarrow Hosts$'. The constraint C_1 can now be modelled as in Example 8. The constraint C_2 could be enforced as follows:

$$\forall (g \in Guests, h \in Hosts)\ \text{count}(0 \ldots 1)(p \in Periods \mid Schedule(g, p) = h)$$

but the same effect can be achieved more succinctly by introducing the *projection* of the *Schedule* function on the guests:

$$\text{where } Schedule(Guests, \_) : Periods \longrightarrow^{0 \ldots 1} Hosts$$

This *projection clause* has to be adjoined to the declaration above of the *Schedule* decision variable, and is automatically compiled into the more complex constraint above. Finally, the constraint C_3 can be captured as follows:

$$\forall (g_1 < g_2 \in Guests)\ \text{count}(0 \ldots 1)(p \in Periods \mid Schedule(g_1, p) = Schedule(g_2, p))$$

Figure 5 shows the resulting pretty-printed ESRA model.

dom $Guests,\ Hosts,\ Periods$
cst $SpareCapacity : Hosts \longrightarrow \mathbb{N},\ CrewSize : Guests \longrightarrow \mathbb{N}$
var $Schedule : (Guests \times Periods) \longrightarrow Hosts$
where $Schedule(Guests, \_) : Periods \longrightarrow^{0\ldots1} Hosts$
solve

$$\forall(p \in Periods,\ h \in Hosts) \left(\sum_{g \in Guests \mid Schedule(g,p)=h} CrewSize(g) \right) \leq SpareCapacity(h)$$

$\wedge$

$$\forall(g_1 < g_2 \in Guests)\ \mathrm{count}(0\ldots1)(p \in Periods \mid Schedule(g_1, p) = Schedule(g_2, p))$$

Fig. 5. A pretty-printed ESRA model for the Progressive Party problem

2.4 Compiling Relational Models

A prototype compiler for ESRA is currently under development. It is being written in OCAML (www.ocaml.org) and compiles an ESRA model into a SICStus Prolog [3] finite-domain constraint program. This choice of the target language is motivated by its excellent collection of global constraints and by our collaboration with its developers on designing new global constraints. We have several statements of interest for developing compilers of ESRA into other target languages. We already have an ESRA-to-OPL compiler for a restriction of ESRA to functions [24, 10]. The 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 these target languages can, and must, be deployed during compilation. In particular, the implementation of decision-variable indices is well-understood.

In order to bootstrap this prototype quickly, we chose the initial simplistic strategy of representing *every* relational variable by a table of 0/1 integer variables, indexed by its participating sets. This 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*. The modeller is then invited to experiment with her (real-life) instances and the resulting compiled programs, so as to determine which one is the 'best'. If the compiler is provided with those instances, then it can be extended to automate such experiments and rankings.

Eventually, more intelligence will be built into the compiler via *heuristics* (such as those of [10]) 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 solver, and this either theoretically (see, e.g., [22] for bijections and [10] for injections) or empirically (see, e.g., [17] for bijections).

Our ultimate aim is of course to design an actual *solver for relational constraints*, without going through compilation. Work in this direction has begun.

3 Benefits of Relational Modelling

In our experience, and as observable 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 a *smaller language* than conventional constraint 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 [10, 17, 22], can be built into the compiler(s), so that even lazy 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, so that even lazy or less competent modellers are well assisted. Model maintenance at the relational level reduces to adapting to the new problem, with all representation (and solving) issues left to the compiler. 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 [15]. If information on the new distribution is given to the 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 because 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, especially using tractability results [21].

There would be further benefits to a relational modelling language if it were adopted as a *standard front-end language* for solvers. Indeed, models and instance data would then be solver-independent and could be shared between solvers. This

would facilitate fair and homogeneous solver comparisons, say via new standard benchmarks, as well as foster competition in fine-tuning the compilers.

4 Conclusion

Summary. We have argued 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; and all this without sacrificing the possibility of efficient solving, so that even lazy 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.

Related Work. We have here generalised and re-engineered our work [6, 24, 10] on a predecessor of ESRA, now called Functional-ESRA, that only supports function variables, by pursuing the plan outlined in [4].

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 [11], which itself gained much from the Z specification notation [18] (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 [20] stands out as a medium-level modelling language and actually gave the impetus to design ESRA: consult [4] for a comparison of elegant ESRA models with more awkward (published) OPL counterparts that do not provide all the benefits of Section 3. Experimental higher-level constraint modelling languages have been proposed, such as ALICE [13], $CLP(Fun(D))$ [9], EACL [19], NCL [25], and NP-SPEC [2]. 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, possibly with functions or relations, while dispensing with such hard-to-properly-implement and rarely-necessary (for constraint modelling) luxuries as recursion, negation, and unbounded quantification. However, ESRA goes beyond them, by advocating an abstract-datatype view of relations, so that their representations need not be fixed in advance, as well as an elegant notation for multiplicity constraints. We lack the space here for a deeper comparison with these languages.

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 have not yet fully worked out the details, but aim at modelling the well-known Travelling Salesperson (TSP) problem as in Figure 6, 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 [13]. As we

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

Fig. 6. A pretty-printed ESRA model for the Travelling Salesperson problem

do not aim at a complete modelling language, we can be very conservative in what missing features shall be added to ESRA when they are identified.

In [14], 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 function and the constraints would need to be textually expressed. Finally, a *search language*, such as SALSA [12] or the one of OPL [20], 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, Maria Fox, Brahim Hnich, Daniel Jackson, François Laburthe, Derek Long, Gerrit Renker, Christian Schulte, and Mark Wallace for stimulating discussions, as well as the constructive reviewers of a previous version of this paper.

References

1. J.-R. Abrial. *The* B*-Book: Assigning Programs to Meanings*. CUP, 1996.
2. M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. In: G. Gupta (ed), *Proc. of PADL'99*, pp. 16–30. LNCS 1551. Springer-Verlag, 1999.
3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. of PLILP'97*, pp. 191–206. LNCS 1292. Springer-Verlag, 1997.
4. P. Flener. Towards relational modelling of combinatorial optimisation problems. In: Ch. Bessière (ed), *Proc. of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001. At www.lirmm.fr/∼bessiere/ws_ijcai01/.
5. 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 Int'l Workshop on Reformulating CSPs*, held at CP'02. At www-users.cs.york.ac.uk/∼frisch/Reformulation/02/.
6. 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.

7. P. Flener, J. Pearson, and M. Ågren. *The Syntax, Semantics, and Type System of* ESRA. At www.it.uu.se/research/group/astra/, April 2003.
8. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.
9. T.J. Hickey. Functional constraints in CLP languages. In: F. Benhamou and A. Colmerauer (eds), *Constraint Logic Programming: Selected Research*, pp. 355–381. The MIT Press, 1993.
10. B. Hnich. *Function Variables for Constraint Programming.* PhD Thesis, Department of Information Science, Uppsala University, 2003. At publications.uu.se/theses/91-506-1650-1/.
11. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. of FSE/ESEC'01*, *Software Engineering Notes* 26(5):62–73, 2001.
12. F. Laburthe and Y. Caseau. SALSA: A language for search algorithms. *Constraints* 7:255–288, 2002.
13. J.-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence* 10(1):29–127, 1978.
14. D. Lesaint. Inferring constraint types in constraint programming. In: P. Van Hentenryck (ed), *Proc. of CP'02*, pp. 462–476. LNCS 2470. Springer-Verlag, 2002.
15. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints* 1(1–2):7–43, 1996.
16. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.
17. B.M. Smith. Modelling a permutation problem. Research Report 18, School of Computing, University of Leeds, UK, 2000.
18. J.M. Spivey. *The* z *Notation: A Reference Manual* (second edition). Prentice, 1992.
19. E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer-aided constraint programming system. In: J. Little (ed), *Proc. of PACLP'99*, pp. 81–93. The Practical Application Company, 1999.
20. P. Van Hentenryck. *The* OPL *Optimization Programming Language.* The MIT Press, 1999.
21. P. Van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Tractable symmetry breaking for CSPs with interchangeable values. In *Proc. of IJCAI'03*, Morgan Kaufmann Publishers, 2003.
22. T. Walsh. Permutation problems and channelling constraints. In *Proc. of LPAR'01*, pp. 377–391. LNCS 2250. Springer-Verlag, 2001.
23. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with* UML. Addison-Wesley, 1999.
24. S. Wrang. *Implementation of the* ESRA *Constraint Modelling Language.* Master's Thesis in Computing Science 223, Department of Information Technology, Uppsala University, Sweden, 2002. At ftp.csd.uu.se/pub/papers/masters-theses/.
25. J. Zhou. Introduction to the constraint language NCL. *J. of Logic Programming* 45(1–3):71–103, 2000.