The Syntax and Semantics of ESRA

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

(Version of March 23, 2001)

1 Introduction

We introduce ESRA, a high-level language for modelling *constraint satisfaction problems* (CSPs), be they *decision problems* (where appropriate values for the problem variables must be found within their domains, subject to some constraints) or *optimisation problems* (where there also is a numeric cost function that has to be optimised).

Starting from the very expressive, declarative, and fast OPL (Optimisation Programming Language) [5], the ESRA language is designed to be even more expressive and declarative, and equally fast [1]. The ESRA language is in fact a superset of a large subset of OPL. Like OPL, the ESRA language is strongly typed, and a sugared version of what is essentially a first-order logic language. Unlike OPL, the ESRA language supports more advanced types, such as mappings, and allows variables of these types as well as of type set, making it an actual set constraint language and thus more expressive than OPL. Unlike in OPL, no procedural search procedures can be given in ESRA models, as they are automatically generated during compilation (see [2] for first ideas on how we plan to achieve this), making ESRA thus also more declarative than OPL. A set of rewrite rules achieves compilation from ESRA into OPL, yielding models that are often very similar to those that a human OPL modeller would (have to) write anyway, so that there is no loss in solving speed compared to (the available search heuristics of) OPL. Consult [1] for some sample ESRA models, and the OPL models they compile into.

Prior knowledge of OPL must be assumed here, as we cannot possibly explain all its used features here. This report is organised as follows. In Section 2, we explain the design decisions behind ESRA. Then, in Section 3, the syntax of ESRA is introduced. In Section 4, the semantics of ESRA is given, by showing how it is compiled into OPL. Finally, in Section 5, we discuss our directions for future work.

2 Design Decisions

In order to design our new modelling language ESRA, we had to take decisions as to its syntax (in Section 2.1), as well as its novel features compared to other modelling languages, such as OPL (in Section 2.2).

2.1 Syntax Decisions

Since OPL is arguably the most expressive constraint programming language available nowadays, we decided to minimise our efforts for compiling ESRA into some executable form by using OPL as target language. Also, since it is arguably not frequently possible to improve on the expressiveness and declarativeness of OPL, a natural choice was to base the syntax of ESRA on the one of OPL, so that entire passages of ESRA programs can be literally copied during compilation. Finally, OPL being a huge language (and our resources being limited), we opted not to cover all of its features; in fact, just like the designers of OPL, we do not really care whether our ESRA language is complete (in any sense) or not, our main driving force being rather the design of a language that is practical for modelling at least some classes of (real-life) CSPs.

As a result, ESRA is a subset of OPL, both because we omitted (for the time being) support of scheduling, floating-point numbers, strings, piecewise linear functions, etc (see Section 3.3 for the complete list), and because we definitively eliminated support of the search primitives, as our objective is to (eventually) generate the (procedural) search part from an analysis of the (declarative) constraint part.

We also do not support a scripting language such as OPL-SCRIPT. However, ESRA also is a superset of that subset, as we introduce useful high-level type constructors and allow the set operators of OPL in actual set constraints. ESRA is thus designed to be more expressive and more declarative than even OPL, without compromising on efficiency (when compared to the available search heuristics of OPL) [2].

These design decisions allow us to benefit, as a side-effect, from OPL's elegantly hiding that it actually is a logic language. Indeed, typed quantifications are replaced by C-like type and variable declarations, conjunction is denoted by a semi-colon (the usual notation in imperative programming for sequential composition), etc. It is unfortunate that plain logic notation is considered repulsive by many programmers, so efforts indeed must be undertaken to give them a language with the look and feel of other languages.

2.2 High-Level Type Constructors

It is generally recognised that the highest-level data structures are:

- *sequences*: element containers where the union operation is associative, with element order and element repetition being relevant;
- *bags* (or *multisets*): sequences where the union operation is commutative, making element order irrelevant;
- *sets*: bags where the union operation is idempotent, making element repetition irrelevant.

Sequences, bags, and sets are of possibly unbounded cardinality. Their usage is recommended for the high-level modelling of problems. At lower levels of abstraction, these data structures can be represented in a variety of ways, using trees, bit vectors, pointers, etc. As we are (here) not interested in sequential access to sequence elements nor in sequences of unbounded cardinality, we abandon sequences in favour of (fixed-size) arrays, with direct access to elements. Similarly, we are (for the time being) not concerned with bags and infinite sets, and ignore them as modelling devices.

So, equipped with (finite) sets and arrays, what can a problem modeller do? More precisely, are there any useful recurring modelling idioms that can be captured in new ways? Following D.R. Smith [4], we claim that many problems are of either of the following four classes: [1]

- *SUBSET*: Find a subset of a given set. For example, finding a clique of a graph amounts to finding a subset of its vertex set.
- *MAPPING*: Find a mapping from a given set to another given set. For example, the colouring of the countries of a map, such that any two neighbour countries have different colours, fits this class.
- *PERMUTATION*: Find an array that represents a permutation of a given set. For example, scheduling jobs according to precedence constraints is a permutation problem.
- *SEQUENCE*: Find an array that represents a sequence, of bounded cardinality, of elements drawn from a given set. For example, a (variant of the) travelling salesperson problem can be modelled this way, with a set of cities being ordered into a route, such that every city is visited at least once.

Going beyond Smith's classification now, we recognise that many real-life problems are actually hybrid in nature, so that we also need to support any *combination* of the four classes above. For instance, the Warehouse Location problem [5] is a hybrid of *SUBSET* and *MAPPING*, because a mapping has to be found from the given set of stores into a subset of the given set of warehouses.

The four classes above are thus actually not classes of stand-alone problems, but rather give rise to powerful high-level *type constructors*, of which several can be used in the same program. The syntax and (informal) semantics of their usage in variable declarations is as follows:

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

```
var {Vertices} Clique;
...
forall(A,B in Clique) <A,B> in Edges;
```

[1]Smith actually identifies seven classes, but we discarded one here, as it is not applicable to CSPs, and we twice merged two of his classes into one.

is the core of a model of the clique problem.

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

  ```
  var Countries->Colours M;
  ...
  forall(A,B in Countries) <A,B> in Neighbours => M[A]<>M[B];
  ```

  is the core of a model of the map colouring problem.

- `var perm(S) A`: Array `A`, indexed by `1..card(S)`, represents a permutation of set `S`. A superset of `S` must be known. For instance:

  ```
  var perm(Jobs) Sched;
  ...
  forall(I,J in 1..card(Sched)) <Sched[I],Sched[J]> in Prec => I<J;
  ```

  is the core of a model of the job scheduling problem.

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

  ```
  int MaxCities = ...;
  var seq(Cities,MaxCities) Route;
  ...
  forall(City in Cities) City in Route;
  ```

  is the core of a model of our travelling salesperson problem.

The *SUBSET* class can be usefully generalised to *nSUBSETS*, where the aim is to find a maximum of n subsets of the given set.

3 The Syntax of ESRA

We now present the syntax of the ESRA language, which is inspired by the one of OPL. But before we can give its BNF grammar, we need to fix some syntactic conventions.

3.1 Syntactic Conventions

To highlight that ESRA is a superset of a subset of OPL, we use the same syntactic conventions to give the grammar of ESRA as used for OPL in [5]:

- Terminal symbols are denoted in typewriter font (e.g., `solve`)
- $\langle nt \rangle$ denotes a non-terminal symbol *nt*
- $\underline{[}$ *object* $\underline{]}$ denotes an optional grammar segment *object*
- $\underline{\{}$ *object* $\underline{\}}$ denotes zero, one, or several times the grammar segment *object*
- *object*$^+$ denotes an expression *object* $\underline{\{}$, *object* $\underline{\}}$
- *object** denotes an expression *object* $\underline{\{}$; *object* $\underline{\}}$

When a nonterminal symbol, say $\langle n \rangle$, is defined by several rules, say as $\langle a \rangle$, $\langle b \rangle$, or $\langle c \rangle$, we use the notation:

$$\begin{array}{lll} \langle n \rangle & \rightarrow & \langle a \rangle \\ & \rightarrow & \langle b \rangle \\ & \rightarrow & \langle c \rangle \end{array}$$

Table 1: Keywords of ESRA

`by`	`card`	`count`	`diff`	`else`
`endif`	`enum`	`exists`	`forall`	`if`
`in`	`int`	`inter`	`max`	`maximize`
`maxint`	`min`	`minimize`	`mod`	`not`
`ordered`	`perm`	`prod`	`range`	`seq`
`solve`	`struct`	`subject`	`subset`	`sum`
`symdiff`	`then`	`to`	`union`	`var`

or the notation:

$$\langle n \rangle \quad \rightarrow \quad \langle a \rangle \mid \langle b \rangle \mid \langle c \rangle$$

depending on convenience in context.

The basic building blocks of ESRA are integers (non-terminal $\langle \mathit{Integer} \rangle$), identifiers (non-terminal $\langle \mathit{Id} \rangle$), and the keywords of the language (e.g., `forall`). Identifiers in ESRA start with a letter and can contain only letters and digits. Note that identifiers in ESRA are *not* case-sensitive (for technical reasons in the compiler design). Also, note that the underscore character '_' is *not* allowed in identifiers (this allows the compiler to easily create new identifiers, when necessary). Integers are sequences of digits, possibly prefixed by a minus sign. The ESRA keywords are listed in Table 1.

3.2 The Grammar of ESRA

ESRA models consist of a set of declarations followed by an instruction:

$$\begin{array}{lll} \langle \mathit{Model} \rangle & \rightarrow & \underline{\{}\ \langle \mathit{Declaration} \rangle\ \underline{\}} \\ & & \langle \mathit{Instruction} \rangle \end{array}$$

Declarations are either type declarations, or instance data declarations, or problem variable declarations:

$$\begin{array}{lll} \langle \mathit{Declaration} \rangle & \rightarrow & \langle \mathit{TypeDecl} \rangle\ \texttt{;} \\ & \rightarrow & \langle \mathit{DataDecl} \rangle\ \texttt{;} \\ & \rightarrow & \langle \mathit{VarDecl} \rangle\ \texttt{;} \end{array}$$

User-declared types can be defined using enumerations (symbol sequences[2] that are initialised either inline, or offline via a data file), integer ranges, and the record (or: structure) type-constructor:

$$\begin{array}{lll} \langle \mathit{TypeDecl} \rangle & \rightarrow & \texttt{enum}\ \langle \mathit{Id} \rangle\ \langle \mathit{EnumInit} \rangle \\ & \rightarrow & \texttt{range}\ \langle \mathit{Id} \rangle\ \texttt{[}\ \langle \mathit{Range} \rangle\ \texttt{]} \\ & \rightarrow & \texttt{struct}\ \langle \mathit{Id} \rangle\ \texttt{\{}\ \langle \mathit{FieldDecl} \rangle^{*}\ \texttt{\}} \\ \langle \mathit{EnumInit} \rangle & \rightarrow & \texttt{\{}\ \langle \mathit{Id} \rangle^{+}\ \texttt{\}} \\ & \rightarrow & \texttt{...} \\ \langle \mathit{Range} \rangle & \rightarrow & \langle \mathit{Expression} \rangle\ \texttt{..}\ \langle \mathit{Expression} \rangle \\ \langle \mathit{FieldDecl} \rangle & \rightarrow & \langle \mathit{Type} \rangle\ \langle \mathit{Id} \rangle\ \underline{[}\ \langle \mathit{Subscripts} \rangle\ \underline{]} \end{array}$$

(Arrays of) instance data can be declared to be of the primitive types (namely integers and non-negative integers), user-declared types (as seen above), as well as set types (built using the set type-constructor).[3] Arrays can have any number of dimensions. Integer ranges, enumerations, and sets can be used to define each dimension. Initialisation of the instance data is either inline, or offline via a data file. Arrays can even be initialised (inline) by generic expressions. The grammar of data declarations is:

[2]In order not to confuse OPL modellers, we uphold the OPL view that enumerations are "totally ordered sets" and thus write their elements between curly braces. In fact, they are symbol sequences without element repetition and can be seen as integer ranges or symbol arrays, so that their elements should be written between square brackets.

[3]In order not to confuse OPL modellers, we uphold the OPL view that instance-data sets are "totally ordered sets" and thus write their elements between curly braces. In fact, they are symbol sequences without element repetition and can be seen as integer ranges or symbol arrays, so that their elements should be written between square brackets.

$$\begin{array}{lll}
\langle \textit{DataDecl} \rangle & \rightarrow & \langle \textit{Type} \rangle\ \langle \textit{Id} \rangle\ \underline{[}\ \langle \textit{Subscripts} \rangle\ \underline{]}\ \langle \textit{DataInit} \rangle \\
\langle \textit{Type} \rangle & \rightarrow & \texttt{int} \mid \texttt{int+} \mid \langle \textit{Id} \rangle \mid \texttt{\{}\ \langle \textit{Type} \rangle\ \texttt{\}} \\
\langle \textit{Subscripts} \rangle & \rightarrow & \texttt{[}\ \langle \textit{Subscript} \rangle^{+}\ \texttt{]} \\
\langle \textit{Subscript} \rangle & \rightarrow & \langle \textit{Range} \rangle \\
 & \rightarrow & \langle \textit{Id} \rangle \\
 & \rightarrow & \langle \textit{Id} \rangle\ \texttt{in}\ \langle \textit{Range} \rangle \\
 & \rightarrow & \langle \textit{Id} \rangle\ \texttt{in}\ \langle \textit{Id} \rangle \\
\langle \textit{DataInit} \rangle & \rightarrow & \texttt{=}\ \langle \textit{ElemInit} \rangle \\
 & \rightarrow & \texttt{= ...} \\
\langle \textit{ElemInit} \rangle & \rightarrow & \text{... see Figure 4.8 p.78 of [5] ...}
\end{array}$$

(Arrays of) problem variables can be declared to be of the primitive types, user-declared types, set types (designating that the set variable is a subset of the set of the given type), as well as some additional types, built using the type constructors `->` (designating all mappings between the two argument sets), `perm` (designating all permutations of the argument set), and `seq` (designating all sequences of elements drawn from the argument set, the size of the sequences being upper-bounded by the value of the argument expression). (Set, mapping, permutation, and sequence variables are not supported by OPL. Set variables represent actual sets, in the mathematical sense, where element order and repetition are irrelevant.) The internal representation of set and mapping variables is hidden from the modeller (and actually context-dependent, as shown in Section 4). The modeller can view permutation and sequence variables as arrays. The grammar of problem variable declarations is:

$$\begin{array}{lll}
\langle \textit{VarDecl} \rangle & \rightarrow & \texttt{var int}\ \langle \textit{Id} \rangle\ \underline{[}\ \langle \textit{Subscripts} \rangle\ \underline{]}\ \underline{[}\ \texttt{in}\ \langle \textit{Range} \rangle\ \underline{]} \\
 & \rightarrow & \texttt{var int+}\ \langle \textit{Id} \rangle\ \underline{[}\ \langle \textit{Subscripts} \rangle\ \underline{]}\ \underline{[}\ \texttt{in}\ \langle \textit{Range} \rangle\ \underline{]} \\
 & \rightarrow & \texttt{var}\ \langle \textit{Id} \rangle\ \langle \textit{Id} \rangle\ \underline{[}\ \langle \textit{Subscripts} \rangle\ \underline{]} \\
 & \rightarrow & \texttt{var}\ \langle \textit{NewType} \rangle\ \langle \textit{Id} \rangle \\
\langle \textit{NewType} \rangle & \rightarrow & \texttt{\{}\ \langle \textit{Id} \rangle\ \texttt{\}} \mid \langle \textit{Id} \rangle\ \texttt{->}\ \langle \textit{Id} \rangle \mid \texttt{perm(}\langle \textit{Id} \rangle\texttt{)} \mid \texttt{seq(}\langle \textit{Id} \rangle\texttt{,}\langle \textit{Expression} \rangle\texttt{)}
\end{array}$$

Expressions on integers, records, enumerations, and sets (the latter two are basically integer ranges) are constructed from constants, instance data, and problem variables, using the traditional (aggregate) operators of arithmetic and sets, as well as parentheses. (Record expressions and the aggregate operator `count` are not supported by OPL.) Operators "/" and `mod` compute the integer quotient and remainder of a division, respectively. The maximum integer representable in ESRA is denoted by constant `maxint`. Operator `card` returns the cardinality of an enumeration or set. Since relations (see below) are seen in ESRA as 0-1 integers (with 0 standing for falsity, and 1 for truth), they can also appear in integer expressions, provided they are wrapped in parentheses. The grammar of expressions is:

$$\begin{array}{lll}
\langle \textit{Expression} \rangle & \rightarrow & \langle \textit{UnOp} \rangle\ \langle \textit{Expression} \rangle \\
 & \rightarrow & \langle \textit{Expression} \rangle\ \langle \textit{BinOp} \rangle\ \langle \textit{Expression} \rangle \\
 & \rightarrow & \langle \textit{AggrOp} \rangle\ \texttt{(}\ \langle \textit{Parameter} \rangle^{+}\ \texttt{)}\ \langle \textit{Expression} \rangle \\
 & \rightarrow & \texttt{count(}\ \langle \textit{Parameter} \rangle^{+}\ \texttt{)} \\
 & \rightarrow & \langle \textit{Integer} \rangle \\
 & \rightarrow & \texttt{maxint} \\
 & \rightarrow & \langle \textit{Argument} \rangle \\
 & \rightarrow & \texttt{(}\ \langle \textit{Expression} \rangle\ \texttt{)} \\
 & \rightarrow & \texttt{(}\ \langle \textit{Relation} \rangle\ \texttt{)} \\
\langle \textit{UnOp} \rangle & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{card} \\
\langle \textit{BinOp} \rangle & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{mod} \mid \texttt{union} \mid \texttt{diff} \mid \texttt{inter} \mid \texttt{symdiff} \\
\langle \textit{AggrOp} \rangle & \rightarrow & \texttt{sum} \mid \texttt{min} \mid \texttt{max} \mid \texttt{union} \mid \texttt{prod} \mid \texttt{inter} \\
\langle \textit{Argument} \rangle & \rightarrow & \langle \textit{Object} \rangle \\
 & \rightarrow & \langle \textit{Id} \rangle\ \langle \textit{Deref} \rangle \\
 & \rightarrow & \langle \textit{Id} \rangle\ \texttt{->}\ \langle \textit{Id} \rangle \\
\langle \textit{Object} \rangle & \rightarrow & \langle \textit{Id} \rangle \\
 & \rightarrow & \texttt{<}\ \langle \textit{Id} \rangle^{+}\ \texttt{>} \\
\langle \textit{Deref} \rangle & \rightarrow & \texttt{[}\ \langle \textit{ElemInit} \rangle^{+}\ \texttt{]} \\
 & \rightarrow & \texttt{.}\ \langle \textit{Id} \rangle
\end{array}$$

Table 2: Operator Precedences in ESRA

Class	Operator	Precedence
logical	`<=>`	0
	`=>`	1
	`\/`	2
	`&`	3
	`not`	4
arithmetic	`=`, `>=`, `<=`, `>`, `<`, `<>`	5
sets	`in`, `not in`, `subset`	5
binary	`+`, `-`, `union`, `diff`, `symdiff`	6
unary	`+`, `-`, `card`	7
aggregate	`sum`, `count`, `min`, `max`, `union`	7
binary	`*`, `/`, `mod`, `inter`	8
aggregate	`prod`, `inter`	9

Relations on integers, sets, and other relations are constructed from expressions using the traditional operators of arithmetic, sets, and logic. Relations are either true or false, and can appear in integer expressions (as seen above). The grammar of relations is:

$$\begin{array}{lcl}
\langle \mathit{Relation} \rangle & \rightarrow & \langle \mathit{Expression} \rangle\ \langle \mathit{ArithOp} \rangle\ \langle \mathit{Expression} \rangle\ \underline{\{}\ \langle \mathit{ArithOp} \rangle\ \langle \mathit{Expression} \rangle\ \underline{\}} \\
 & \rightarrow & \langle \mathit{Expression} \rangle\ \langle \mathit{SetOp} \rangle\ \langle \mathit{Expression} \rangle \\
 & \rightarrow & \texttt{not}\ \langle \mathit{Relation} \rangle \\
 & \rightarrow & \langle \mathit{Relation} \rangle\ \langle \mathit{LogicOp} \rangle\ \langle \mathit{Relation} \rangle \\
\langle \mathit{ArithOp} \rangle & \rightarrow & \texttt{=} \mid \texttt{>=} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{<} \mid \texttt{<>} \\
\langle \mathit{SetOp} \rangle & \rightarrow & \texttt{in} \mid \texttt{not in} \mid \texttt{subset} \\
\langle \mathit{LogicOp} \rangle & \rightarrow & \texttt{\&} \mid \texttt{\textbackslash/} \mid \texttt{=>} \mid \texttt{<=>}
\end{array}$$

(Conjunctions of) constraints on the problem variables are posted using relations (as above), the traditional aggregate operators (or: quantifiers) of logic, and conditional composition. (The existential quantifier `exists` is not supported by OPL.) The grammar of constraints is:

$$\begin{array}{lcl}
\langle \mathit{Constraint} \rangle & \rightarrow & \langle \mathit{Relation} \rangle \\
 & \rightarrow & \texttt{exists (}\ \langle \mathit{Parameter} \rangle^{+}\ \texttt{)}\ \langle \mathit{Constraint} \rangle \\
 & \rightarrow & \texttt{forall (}\ \langle \mathit{Parameter} \rangle^{+}\ \texttt{)}\ \langle \mathit{Constraint} \rangle \\
 & \rightarrow & \texttt{if}\ \langle \mathit{Relation} \rangle\ \texttt{then}\ \langle \mathit{Constraint} \rangle\ \underline{[}\ \texttt{else}\ \langle \mathit{Constraint} \rangle\ \underline{]}\ \texttt{endif} \\
 & \rightarrow & \{\ \langle \mathit{Constraint} \rangle^{*}\ \}
\end{array}$$

Formal parameters are needed for all aggregate operators. They must be within some bounds (an integer range, enumeration, or set), and may be required to satisfy some additional condition; the common condition that they be ordered under "<" (recall that enumerations and sets are basically integer ranges), can be more simply expressed with the `ordered` keyword. The grammar of formal parameters is:

$$\begin{array}{lcl}
\langle \mathit{Parameter} \rangle & \rightarrow & \underline{[}\ \texttt{ordered}\ \underline{]}\ \langle \mathit{Object} \rangle^{+}\ \texttt{in}\ \langle \mathit{Bounds} \rangle\ \underline{[}\ \texttt{:}\ \langle \mathit{Relation} \rangle\ \underline{]} \\
\langle \mathit{Bounds} \rangle & \rightarrow & \langle \mathit{Argument} \rangle \mid \langle \mathit{Range} \rangle
\end{array}$$

The instruction of an ESRA model posts the constraints of the problem, and states the optional cost function that has to be optimised. The grammar of instructions is:

$$\begin{array}{lcl}
\langle \mathit{Instruction} \rangle & \rightarrow & \texttt{solve}\ \langle \mathit{Constraint} \rangle\ \texttt{;} \\
 & \rightarrow & \texttt{minimize}\ \langle \mathit{Expression} \rangle\ \texttt{subject to}\ \langle \mathit{Constraint} \rangle\ \texttt{;} \\
 & \rightarrow & \texttt{maximize}\ \langle \mathit{Expression} \rangle\ \texttt{subject to}\ \langle \mathit{Constraint} \rangle\ \texttt{;}
\end{array}$$

The precedence of all operators is given in Table 2.

3.3 Features of OPL Not Supported by ESRA

So far, we have shown all the features of ESRA, stating whether they are taken verbatim from OPL, or enhanced over their definition in OPL, or new compared to OPL. The features of OPL that are not supported in ESRA are divided into two categories. First, the following OPL features are *currently* not supported in ESRA, because of our limited resources (figure and page numbers refer to [5], which is about OPL version 2):

- floating point numbers, ranges, instance data, problem variables, and expressions (in Figure 4.2 p.68, in Figure 4.3 p.69, in Figure 4.4 p.71, and in Figure 5.2 p.95)
- scheduling declarations, operations, constraints, and reflective functions (Figure 4.5 p.71, in Figure 5.3 p.96, in Figure 5.4 p.97, and Figure 11.2 p.217)
- constraint declarations for naming individual constraints (Figure 4.6 p.76)
- constraint assertions for checking consistency of the instance data (Figure 4.7 p.76)
- file initialisations for a *single* instance datum (in Figure 4.8 p.78)
- offline initialisations via a `data` instruction (Figure 4.9 p.80)
- computed initialisations via an `Initialize` instruction (Figure 4.10 p.82)
- piecewise linear functions (in Figure 5.2 p.95)
- display instructions for pretty-printing results (in Figure 8.1 p.140)
- predicate declarations for defining (a limited version of) procedures (new in OPL version 3, in Figure 5.5 of its reference manual)
- setting declarations (new in OPL version 3, in Figure 9.1 of its reference manual)
- database operations (new in OPL version 3, in Figure 10.1 of its reference manual)
- scripting language (new with OPL version 3, in its reference manual)

Second, the following other OPL features will *never* be supported in ESRA, because we (plan to) generate (during compilation from ESRA into OPL) code that uses these often rather procedural features (figure and page numbers refer to [5]):

- reflective functions giving information about the current state of the computation (Table 5.1 p.93)
- global constraints (in Figure 5.4 p.97)
- forcing the use of linear relaxations of integer constraints (in Figure 5.5 p.107)
- search procedures encoding labelling heuristics (Figure 7.1 p.122)

New features of future versions of OPL will undergo similar classification.

4 The Semantics of ESRA

We now explain the semantics of the ESRA language, by exhibiting the architecture of a compiler (into OPL), and showing that the main modules of that architecture can be easily implemented by a set of ESRA-to-OPL rewrite rules.

4.1 Architecture of our ESRA Compiler

The architecture of our ESRA compiler is shown in Figure 1. First, the Decomposer separates an ESRA program into its declaration, optimisation, and constraint parts. Next, the ESRA-to-OPL Declaration Converter rewrites all ESRA declarations into OPL declarations, and possibly into some OPL constraints and OPL display statements (see Section 4.2.1 for details). Also, the ESRA-to-OPL Converter rewrites the ESRA optimisation and constraint parts into an OPL optimisation part and more OPL constraints, using the declaration part (again see Section 4.2.1 for details). Finally, the Composer assembles the generated OPL program by suitably concatenating the obtained OPL statements.

The Decomposer and Composer modules are trivial, and are not discussed here. The converter modules are explained next.

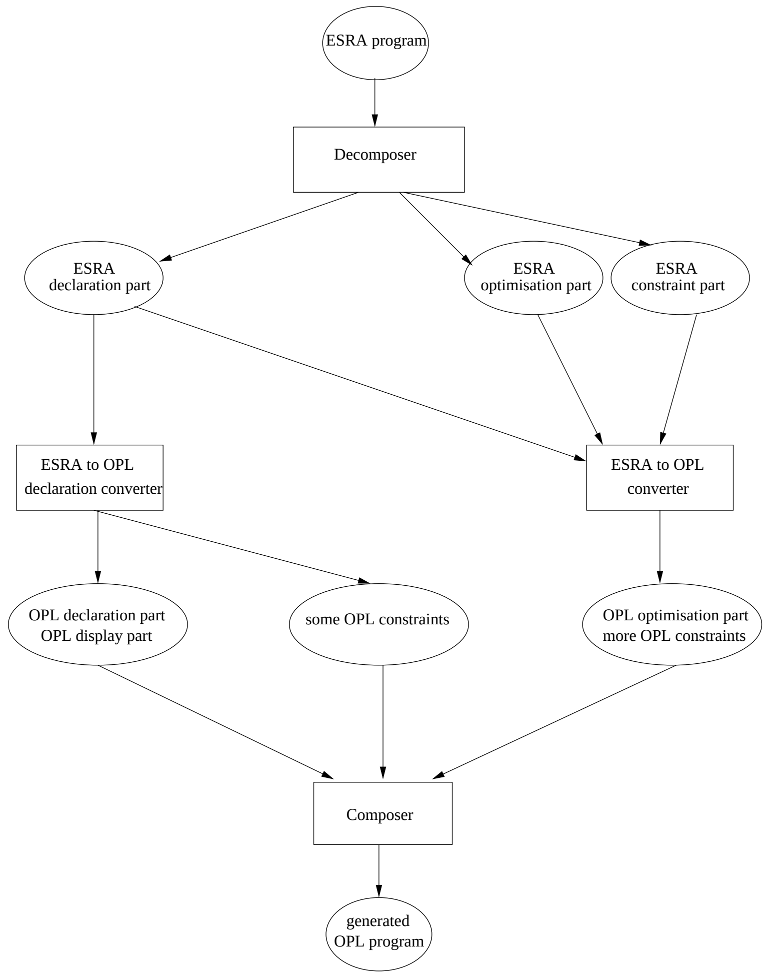

Figure 1: Architecture of our ESRA compiler

4.2 ESRA-to-OPL Rewrite Rules

We use conditional rewrite rules, here written as follows:

$$L \;\Rightarrow\; R \quad | \quad C$$

meaning that, if condition C holds, then expression L is rewritten into R.

4.2.1 ESRA-to-OPL Declaration Converter

The declarations of ESRA that involve types not supported (in the same way) by OPL (namely sets, mappings, permutations, and sequences) are rewritten into OPL declarations, and possibly into some OPL constraints and display statements. All other declarations literally become OPL declarations. For each variable declaration involving n sets, there are 2^n rewrite rules, depending on whether each set is itself a variable or not. No instance data that are mappings, permutations, or sequences are allowed. Furthermore, all new variable names introduced by the rewrite rules have an underscore character, which makes sure that they were not used by the modeller, since ESRA variable names are not allowed to have an underscore.

Set rules. For set declarations, we have two rules.

- The first rule is as follows:

```
var {T} S;
  ⇒   var int S[T] in 0..1;
      display(I in T: S[I]=1) <I>;
  |   T is a known set
```

A set S of known super-set T is thus represented as an array of Boolean variables, indexed by T. Furthermore, a display statement is generated to pretty-print S.

- The second rule is:

```
var {T} S;
  ⇒   var int S[U] in 0..1;
      forall(I in U) S[I]=1 => T[I]=1;
      display(I in U: S[I]=1) <I>;
  |   T is a set variable of known super-set U
```

A set S of variable super-set T of known super-set U is thus represented as an array of Boolean variables, indexed by U. A constraint is added to force elements from S to be also in T. Furthermore, a display statement is generated to pretty-print S.

Mapping rules. We have four rules for mapping variable declarations.

- The first rule is:

```
var V->W M;
 ⇒  var W M[V];
  |   V and W are known sets
```

The mapping M is thus represented as an array of variables drawn from W, indexed by V.

- The second rule is:

```
var V->W M;
  ⇒   var T M[V];
      forall(I in V) W[M[I]]=1;
  |   V is a known set, and W is a set variable of known super-set T
```

The mapping M is thus represented as an array of variables drawn from T, indexed by V. Furthermore, we post the constraint that every actually mapped element of T must be a member of W.

- The third rule is:

```
var V->W M;
  ⇒   var W M[S];
      display(I in S: V[I]=1) <I,M[I]>;
  |   V is a set variable of known super-set S, and W is a known set
```

The mapping M is thus represented as an array of variables drawn from W, indexed by S. Furthermore, a display statement is generated to pretty-print M.

- The fourth rule is:

```
var V->W M;
  ⇒   var int M[S,T] in 0..1;
      forall(I in S, J in T) M[I,J]=1 => V[I]=1 & W[J]=1;
      forall(I in S) V[I]=1 => (sum(J in T) (M[I,J]=1))=1;
      display(I in S, J in T: M[I,J]=1) <I,J>;
  |   V and W are set variables of known super-sets S and T, respectively
```

The mapping M is thus represented as a two-dimensional array of Booleans, indexed by S and T. Furthermore, we post the constraint that every actual pair <I,J> of M forces I and J to be members of V and W, respectively. Also, we post the constraint that every element of V must be mapped to exactly one element of W, because modelling the mapping as a Boolean matrix does not by itself enforce this. Finally, a display statement is generated to pretty-print M.

Permutation rules. We have two rules for permutation variable declarations. A permutation of a set `V` is viewed as an array, indexed by `1..card(V)`, so that modellers have direct access to its elements.

- The first rule is:

```
var perm(V) P;
  ⇒    var V P[1..card(V)];
       alldifferent(P);
  |    V is a known set
```

The permutation `P` is thus represented as an array of variables drawn from `V`, indexed by `1..card(V)`. Furthermore, we post the constraint that all elements of `P` must be different.

- The second rule is:

```
var perm(V) P;
  ⇒    var T P[1..card(T)];
       var int P_s[1..card(T)] in 0..1;
       alldifferent(P);
       forall(I in 1..card(T)) P_s[I]=1 <=> V[P[I]]=1;
       forall(I in 2..card(T)) P_s[I-1]>=P_s[I];
       display(I in 1..card(T): P_s[I]=1) <P[I]>;
  |    V is a set variable of known super-set T
```

The permutation `P` is thus represented as an array of variables drawn from `T`, indexed by `1..card(T)`. Furthermore, we post the constraint that all elements of `P` must be different. We also create an array of Booleans, `P_s`, indexed by `1..card(T)`, to denote the subset of the permutation to be computed. We force that every element in `P` must also be in the set variable `V`. We also add a symmetry-breaking constraint in the form of inequalities between the Booleans of `P_s` that force `P` to have as prefix all the actual elements in the permutation and as a suffix all the elements that are excluded from the permutation. Furthermore, a display statement is generated to pretty-print `P`.

Sequence rules. We have two rules for sequence variable declarations. A sequence of bounded length `K` over a set `V` is viewed as an array, indexed by `1..K`, so that modellers have direct access to its elements.

- The first rule is:

```
var seq(V,K) S;
  ⇒    var V S[1..K];
       var int S_s[1..K] in 0..1;
       forall(I in 2..K) S_s[I-1]>=S_s[I];
       display(I in 1..K: S_s[I]=1) <S[I]>;
  |    V is a known set
```

The sequence `S` is thus represented as an array of variables drawn from `V`, indexed by `1..K`. We also create an array of Booleans, `S_s`, indexed by `1..K` to denote the subsequence to be computed of the sequence. We add a symmetry-breaking constraint in the form of inequalities between the Booleans of `S_s` that force `S` to have as prefix all the actual elements in the sequence and as a suffix all the elements that are excluded from the sequence. Furthermore, a display statement is generated to pretty-print `S`.

- The second rule is:

```
var seq(V,K) S;
  ⇒    var T S[1..K];
       var int S_s[1..K] in 0..1;
       forall(I in 1..K) S_s[I]=1 <=> V[S[I]]=1;
       forall(I in 2..K) S_s[I-1]>=S_s[I];
       display(I in 1..K: S_s[I]=1) <S[I]>;
  |    V is a set variable of known super-set T
```

The sequence S is thus represented as an array of variables drawn from T, indexed by 1..K. We also create an array of Booleans, S_s, indexed by 1..K, to denote the subsequence to be computed of the sequence. We force every element in S to be in the variable set V. We also add a symmetry-breaking constraint in the form of inequalities between the Booleans of S_s that force S to have as prefix all the actual elements in the sequence and as a suffix all the elements that are excluded from the sequence. Furthermore, a display statement is generated to pretty-print S.

4.2.2 ESRA-to-OPL Converter

Expressions and constraints of ESRA that involve types not supported (in the same way) by OPL (namely sets, mappings, permutations, and sequences) are rewritten into OPL. Similarly for the expressions and constraints of ESRA that do not exist in OPL (such as `count`). For each ESRA constraint or expression involving n sets (provided that they are not all known, because otherwise it is simply an OPL constraint or expression), there are $2^n - 1$ rewrite rules, depending on whether each set is itself a variable or not.

Subset rules. There are three rules for the `subset` constraint.

- The first rule is:

```
S subset T;
  ⇒   forall(e in S) T[e]=1;
       S subset W;
  |    S is a known set, and T is a variable set of known super-set W
```

  because ...

- The second rule is:

```
S subset T;
 ⇒  forall(I in V diff T) S[I]=0;
 |   S is a set variable of known super-set V, and T is a known set
```

  because ...

- The third rule is:

```
S subset T;
  ⇒   forall(I in V diff W) S[I]=0;
       forall(I in W inter V) S[I]=1 => T[I]=1;
  |    S and T are variable sets of known super-sets V and W, respectively
```

  because ...

Intersection rules. For the `inter` operator, we only allow constraints of the form `A inter B = C`. Similarly for `union`, `diff`, and `symdiff`. There are seven rules for `inter`.

- The first rule is:

```
S inter T = U
  ⇒   U subset (V inter T);
       forall(e in U) S[e]=1;
       forall(e in (V inter T) diff U) S[e]=0;
  |    S is a set variable of known super-set V,
       T is a known set, and
       U is a known set
```

  because ...

- The second rule is:

```
S inter T = U
   ⇒   U subset (S inter W);
       forall(e in U) T[e]=1;
       forall(e in (W inter S) diff U) T[e]=0;
   |   S is a known set,
       T is a set variable of known super-set W, and
       U is a known set
```

because ...

- The third rule is:

```
S inter T = U
   ⇒   (S inter T) subset X;
       forall(e in S inter T) U[e]=1;
       forall(e in X diff (S inter T)) U[e]=0;
   |   S is a known set,
       T is a known set, and
       U is a set variable of known super-set X
```

because ...

- The fourth rule is:

```
S inter T = U
   ⇒   U subset (V inter W);
       forall(e in U) S[e]=1;
       forall(e in U) T[e]=1;
       forall(e in (V inter W) diff U) S[e]=0;
       forall(e in (V inter W) diff U) T[e]=0;
   |   S is a set variable of known super-set V,
       T is a set variable of known super-set W, and
       U is a known set
```

because ...

- The fifth rule is:

```
S inter T = U
   ⇒   forall(e in (V inter T) diff (V inter T inter X)) S[e]=0;
       forall(e in X diff (V inter T inter X)) U[e]=0;
       forall(e in V inter T inter X) S[e]=1 <=> U[e]=1;
       forall(e in (X inter V) diff (V inter T inter X)) S[e]=0;
   |   S is a set variable of known super-set V,
       T is a known set, and
       U is a set variable of known super-set X
```

because ...

- The sixth rule is:

```
S inter T = U
   ⇒   forall(e in (S inter W) diff (S inter W inter X)) T[e]=0;
       forall(e in X diff (S inter W inter X)) U[e]=0;
       forall(e in S inter W inter X) T[e]=1 <=> U[e]=1;
       forall(e in (X inter W) diff (S inter W inter X)) T[e]=0;
   |   S is a known set,
       T is a set variable of known super-set W, and
       U is a set variable of known super-set X
```

because ...

- The seventh rule is:

```
S inter T = U
  ⇒   forall(e in (V inter W) diff (V inter W inter X)) S[e]=0;
      forall(e in (V inter W) diff (V inter W inter X)) T[e]=0;
      forall(e in X diff (V inter W inter X)) U[e]=0;
      forall(e in V inter W inter X) S[e]=1 <=> U[e]=1;
      forall(e in V inter W inter X) T[e]=1 <=> U[e]=1;
      forall(e in (X inter V) diff (V inter W inter X)) S[e]=0;
      forall(e in (X inter W) diff (V inter W inter X)) T[e]=0;
  |   S is a set variable of known super-set V,
      T is a set variable of known super-set W, and
      U is a set variable of known super-set X
```

  because ...

The rules for `union`, `diff`, and `symmdiff` are omitted in this (version of this) report.

Sums over sets. There is only one rule for sums over sets, namely:

```
sum(I in S) F(I)
  ⇒   sum(I in V) F(I)*S[I]
  |   S is a set variable of known super-set V, and F is any ESRA numeric expression
```

because, in this case, the set variable `S` is represented by an array of Booleans, indexed by `V`.

Sums over mappings. We have four rules for sums over mappings.

- The first rule is:

```
sum(I->J in M) F(I,J)
  ⇒   sum(I in S) F(I,M[I])
  |   S and T are known sets,
      M is a mapping from S into T, and
      F is any ESRA numeric expression
```

  because, in this case, the mapping `M` is represented by an array of elements drawn from `T`, indexed by `S`.

- The second rule is:

```
sum(I->J in M) F(I,J)
  ⇒   sum(I in S) F(I,M[I])
  |   S is a known set,
      T is a set variable of known super-set W,
      M is a mapping from S into T, and
      F is any ESRA numeric expression
```

  because, in this case, the mapping `M` is represented by an array of elements drawn from `W`, indexed by `S`.

- The third rule is:

```
sum(I->J in M) F(I,J)
  ⇒   sum(I in V) F(I,M[I])*S[I]
  |   S is a set variable of known super-set V,
      T is a known set,
      M is a mapping from S into T, and
      F is any ESRA numeric expression
```

  because, in this case, the mapping `M` is represented by an array of elements drawn from `T`, indexed by `V`, and the set variable `S` is represented by an array of Booleans, indexed by `V`.

- The fourth rule is:

```
sum(I->J in M) F(I,J)
  ⇒   sum(I in V, J in W) F(I,J)*M[I,J]
  |   S is a set variable of known super-set V,
      T is a set variable of known super-set W,
      M is a mapping from S into T, and
      F is any ESRA numeric expression
```

because, in this case, the mapping `M` is represented by an array of Booleans, indexed by `V` and `W`.

Sums over permutations. The only is one rule for sums over permutations, namely:

```
sum(I in P) F(I)
  ⇒   sum(I in 1..card(W)) F(P[I])*P_s[I]
  |   P is a permutation of a set variable V of known super-set W, and
      F is any ESRA numeric expression
```

because, in this case, `P` is represented by an array of elements drawn from `W`, indexed by `1..card(V)`, and by `P_s` (an array of Booleans controlling the inclusion of some elements in the permutation).

Sums over sequences. There only is one rule for sums over sequences, namely:

```
sum(I in S) F(I)
  ⇒   sum(I in 1..K) F(S[I])*S_s[I]
  |   S is a sequence of bounded size primK of a set variable V, and
      F is any ESRA numeric expression
```

because, in this case, the set variable `S` is represented by an array of elements drawn from `V`, indexed by `1..K`, and by `S_s` (an array of Booleans controlling the inclusion of some elements in the sequence).

Products, minima, maxima, unions, and intersections. The rules for `prod`, `min`, `max`, (n-ary) `union`, and (n-ary) `inter` are omitted in this (version of this) report.

Cardinality of sets. There only is one rule for the cardinality of sets, namely:

```
card(S)
  ⇒  sum(I in V) S[I]
  |  S is a set variable of known super-set V
```

because, in this case, the set `S` is represented by a Boolean array, indexed by `V`.

Cardinality of permutations. There only is one rule for the cardinality of permutations, namely:

```
card(P)
  ⇒  sum(I in 1..card(V)) P_s[I]
  |  P is a permutation of a set variable of known super-set V
```

because, in this case, the permutation `P` is represented by an array of elements drawn from `V`, indexed by `1..card(V)`, and by `P_s` (an array of Booleans controlling the inclusion of some elements in the permutation).

Cardinality of sequences. There only is one rule for the cardinality of sequences, namely:

```
card(S)
  ⇒  sum(I in 1..K) S_s[I]
  |  S is a sequence of bounded size K
```

because, in this case, the sequence `S` is represented by an array indexed by `1..K` and by `S_s` (an array of Booleans controlling the inclusion of some elements in the sequence).

Membership in sets. There only is one rule for membership in sets, namely:

```
I in S
  ⇒  S[I] = 1; I in V
  |  S is a set variable of known super-set V
```

because in this case, `S` is represented by a Boolean array, indexed by `V`. We also need to make sure that `I` is an element of `V`.

Membership in mappings. There are four rules for membership in mappings.

- The first rule is:

```
I->J in M
   ⇒    M[I]=J
   |    V and W are known sets, and
        M is a mapping from V into W
```

  because, in this case, the mapping `M` is represented by an array of elements drawn from `W`, indexed by `V`.

- The second rule is:

```
I->J in M
   ⇒    M[I]=J
   |    V is a known set,
        W is a set variable of known super-set T, and
        M is a mapping from V into W
```

  because, in this case, the mapping `M` is represented by an array of elements drawn from `T`, indexed by `V`, and the set `W` is represented by a Boolean array, indexed by `T`.

- The third rule is:

```
I->J in M
   ⇒    M[I]=J
   |    V is a set variable of known super-set T,
        W is a known set, and
        M is a mapping from V into W
```

  because, in this case, the mapping `M` is represented by an array of elements drawn from `W`, indexed by `T`, and the set `V` is represented by a Boolean array, indexed by `T`.

- The fourth rule is:

```
I->J in M
   ⇒    M[I,J]=1
   |    V is a set variable of known super-set S,
        W is a set variable of known super-set T, and
        M is a mapping from V into W
```

  because, in this case, the mapping `M` is represented by an array of booleans, indexed by `T` and `S`. The set `V` is represented by a Boolean array, indexed by `S`. The set `W` is represented by a Boolean array, indexed by `T`.

Membership in permutations. There are two rules for membership in permutations.

- The first rule is:

```
I in P
  ⇒  I in V
  |  P is a permutation of known super-set V
```

because in this case, `P` is represented by an array of elements drawn from `V`. We only need to make sure that `I` is an element of `V`.

- The second rule is:

```
I in P
 ⇒  I in V; (sum(J in 1..card(V)) P[J]=I*P_s[J]) = 1
  |  P is a permutation of a set variable of known super-set V
```

because in this case, `P` is represented by an array of elements drawn from `T`, indexed by `1..card(V)`. We need to make sure that `I` is an element of `V` as well as being an element of `S`.

Membership in sequences. There only is one rule for membership in sequences, namely:

```
I in S
 ⇒  I in T; (sum(J in 1..K) S[J]=I*S_s[J]) >= 1
  |  S is a sequence of a set variable (of known super-set T) or a known set T
```

because in this case, `S` is represented by an array of elements drawn from `T`, indexed by `1..K`. We need to make sure that `I` is an element of `T` as well as being an element of `S`.

Non-membership. One of the rules for non-membership in sequences is as follows:

```
I not in S
 ⇒  not(I in T & (sum(J in 1..K) S[J]=I * S_s[J]) >= 1)
  |  S is a sequence of a known set T
```

because in this case, `S` is represented by an array of elements drawn from `T`, indexed by `1..K`. We only need to post the negation of the constraint for membership.

The other rules for non-membership can be done in a similar way, and are skipped in this (version of this) report.

Count expressions. A count expression `count(I in S: C)` over any data structure `S`, where `C` is any ESRA numeric expression, can be rewritten (within ESRA) into `sum(I in S) C`, so that we can then use the rules for `sum` discussed previously.

Universal quantification. For universal quantification over sets, one of the rules is as follows:

```
forall(I in S) F(I);
 ⇒  forall(I in V) S[I]=1 => F(I);
  |  S is a set variable of known super-set V, and F is any ESRA constraint
```

because, in this case, the set `S` is represented by a Boolean array, indexed by `V`. In fact, this rule is very similar to the one of the summation rules (discussed previously) except that `S[I]` (in the summation) is replaced by `S[I]=1` (in the quantification), and that the multiplication is replaced with an implication. Hence, the other rules for universal quantification over sets, mappings, permutations, and sequences are skipped in this report.

Existential quantification. An existential quantification `exists(X in S) F(X)` over any data structure `S`, where `F` is any ESRA relation, can be rewritten (within ESRA) into `(sum(X in S) (F(X))) > 0`, so that we can then use the rules for `sum` discussed previously.

5 Future Work

In order to fulfill our design intention of making ESRA more declarative than OPL, whereby we omitted search parts from the ESRA language, we are currently investigating the compile-time generation of also a (procedural) OPL search part from a (declarative) ESRA model. See [2, 3] for first results.

We will also study the reformulation of ESRA models, by investigating the merits of alternative OPL representations of high-level data structures, considering the integration of alternative models, and examining the generation of implied constraints.

Acknowledgements

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

References

[1] 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.

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

[3] Z. Kızıltan, P. Flener, and B. Hnich. A labelling heuristic for subset problems. Submitted for review. Available via http://www.dis.uu.se/~pierref/astra/.

[4] D.R. Smith. The structure and design of global search algorithms. *Tech. Rep. KES.U.87.12*, Kestrel Institute, 1988.

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