The Essence of ESSENCE:

A Constraint Language for Specifying Combinatorial Problems

Alan M. Frisch[1], Matthew Grum[1], Chris Jefferson[1],
Bernadette Martínez Hernández[1], and Ian Miguel[2]

[1] Artificial Intelligence Group, Dept. of Computer Science, Univ. of York, York, UK
[2] School of Computer Science, University of St Andrews, St Andrews, UK

Abstract. ESSENCE is a new language for specifying combinatorial (decision or optimisation) problems at a high level of abstraction. The key feature enabling this abstraction is the provision of decision variables whose values can be combinatorial objects, such as tuples, sets, multisets, relations, partitions and functions. ESSENCE also allows these combinatorial objects to be nested to arbitrary depth, thus providing, for example, sets of partitions, sets of sets of partitions, and so forth.

1 Introduction

This paper describes ESSENCE, a new language for specifying combinatorial (decision or optimisation) problems at a high level of abstraction. ESSENCE is the result of our attempt to design a formal language that enables problem specifications that are similar to rigorous natural-language specifications, such as those catalogued by Garey and Johnson [1]. Formal problem specifications could facilitate communication between humans better than the informal specifications that are currently used, such as those in CSPLib.

A related goal in designing ESSENCE has been to allow problems to be specified at a level of abstraction above that at which decisions are made when modelling the problem in a constraint language, a mathematical programming language or via a SAT encoding. Our principal motivation for a language is to formalise the modelling process: modelling is the transformation of an ESSENCE specification into an equivalent specification in the modelling language of choice. This, in turn, has enabled us to work on the automation of modelling.

Our working hypothesis has been that a problem specification language would not be some form of logical language, such as Z or NP-SPEC, but would be, to a first approximation, a constraint language, such as OPL, $\mathcal{F}$ or ESRA, enhanced with features that increase its level of abstraction. Most importantly, as a combinatorial problem requires finding a certain type of combinatorial object, the language should have decision variables whose domain elements are combinatorial objects of that type. This enables problems to be stated directly and naturally; without the decision variables of the appropriate type the problem would have to be "modelled" by encoding the desired combinatorial object as a collection of constrainted decision variables of some other type.

Constraint programming languages have gradually evolved a greater range of types for decision variables. For example, Eclipse [2] supports decision variables

whose domain elements are sets; similarly $\mathcal{F}$ [3] supports functions, ESRA [4] supports relations and functions, and NP-Spec [5] supports sets, permutations, partitions and integer functions. Each increase in abstraction allows the user to ignore additional modelling decisions, leaving this to the computer. ESSENCE makes a large leap in this direction by providing a type system for constructing decision variables of arbitarily-complex domains. This, and other features, gives ESSENCE a high level of abstraction—enough abstraction that we call ESSENCE a specification language as opposed to a modelling language.

One reason why new types of decision variables have been incorporated into constaint programming languages so slowly has been the difficulty of implementing the enhancements. Elsewhere we have shown how this difficulty can be overcome. In particular, we have shown how ESSENCE specifications can be translated—we say refined—into declarative models at the level of abstraction supported by existing constraint programming languages [6]. We have implemented a rule-based system, called CONJURE, that can refine specifications in a fragment of ESSENCE that contains all of the main characteristics of ESSENCE, but only a few constructs of the language. We believe that all the techniques needed to refine the full language are present in the existing CONJURE implementation. Thus, our development of ESSENCE has been untethered by the demands of refinement; in no case have we omitted a feature or construct from ESSENCE because we could not refine it. Nonetheless, the demands of effective refinement have influenced some of the finer decisions in designing ESSENCE.

The high level of abstraction provided by ESSENCE is primarily a consequence of four features. (1) The language supports a wide range of types (including sets, multisets, relations, functions and partitions) and decision variables can have domains containing values of any one of these types. For example, the Social Golfers Problem (SGP, [7]) requires partitioning a set of golfers into groups (e.g., foursomes) in each week of play subject to a certain constraint. The partitions are easily represented by decision variable whose domain elements are of type partition. We say that the type of a decision variable is the type of its domain elements. (2) All the types can be nested to arbitrary depth; for example a decision variable can be of type set, set of sets, set of sets of sets, and so forth. For example, the SONET problem [7] requires placing each of a set of communicating nodes onto one or more communication rings in such a way that the specified communication demand is met. Thus, the goal is to find a set of rings, each of which is set of nodes—and this can be stated in ESSENCE by using a decision variable of type set of sets. (3) Constraints can contain quantifiers that range over decision variables. For example, if a decision variable X is of type set of sets, a constraint can be of the form $\forall x \in X.\phi$. The Golomb Ruler Problem (GRP, [7]) requires finding a set of integers such that no two distinct pairs of distinct elements have the same difference. This can be expressed directly in ESSENCE by quantifying over the pairs in the set, even though the set is unknown. (4) The language provides types containing unnamed, indistinguishable elements. This is useful because many problems involve some set of elements, yet do not mention particular elements. A guiding design principle is that the language should not force a specification to provide unnecessary information. We know of no specifi-

cation or model of the SGP in which the constraints name any particular golfer. Yet most models name the golfers and, in doing so, introduce symmetry into the model. These four features are almost unique to ESSENCE — ESRA and F support quantifiers ranging over decision variables.

We present a truth-conditional semantics for ESSENCE, a first (we believe) for a language of this kind.[1] It features a proper treatment of undefined expressions, which can arise from, for example, division by zero or array indices out of bounds.

2 An Introduction to ESSENCE by Example

Let us begin by considering the specification of the Golomb Ruler Problem (GRP, problem 6 at www.csplib.org, shown in Fig. 1). A specification is a list of statements, of which there are seven kinds, signalled by the keywords `given`, `where`, `letting`, `find`, `maximising`, `minimising` and `such that`. Statements are composed into specifications according to the regular expression:

(`given` | `letting` | `where`)* `find`$^+$ [`minimising` | `maximising`] (`such that`)*

`letting` statements declare constant identifiers and user-defined types. `given` statements declare problem parameters, whose values are input to specify the instance of the problem class. As in other modelling languages, parameter values are not part of the problem specification. `where` statements are used to constrain parameter values; only valid parameter values specify a problem instance. `find` statements declare decision variables. A `minimising` or `maximising` statement gives the objective function, if any. Finally, `such that` statements give the problem's constraints.

The GRP specification begins by declaring the parameter n (valid when positive) and the identifier *bound*. The declaration of *bound* uses n, so n must be declared first. Identifiers must be declared before use, preventing cyclical definitions and decision variables from being used to define constants or parameters.

ESSENCE is a statically typed language. It supports the atomic types `int` (integer), `bool` (Boolean), user-defined enumerated types and user-defined unnamed types, the last denoted by `type of size` α, a type comprising α unnamed and indistinguishable elements. Compound types are built with the constructors sets, multisets, functions, tuples, relations, partitions and matrices. For example, `set of int` and `relation on int` $\times$ `int` are both types.

The *domain* of a decision variable is a finite set of elements all of the same type. One can specify a domain merely by giving its type, in which case the domain consists of all elements of that type. For example, given the user-defined enumerated type *colour* consisting of the elements *red*, *green* and *blue* one can specify a decision variable with the domain *colour*. One can also specify a domain by giving both a type and *restrictions* that select a subset of the elements of the type. Consider, for example, the GRP decision variable *Ticks*. Its domain is `set of int`, with the restrictions that the set must have cardinality n, and its elements

---

[1] Constraint logic programming languages' semantics focus on integrating constraints with logic programming and abstract away from semantics of particular constraints.

Given n, put n integer ticks on a ruler of size m such that all inter-tick distances are unique. Minimise m.

```
given      n : int
where      n ≥ 0
letting    bound be 2^n
find       Ticks : set (size n) of int (0..bound)
minimising max(Ticks)
such that  ∀_{i,j}⊆Ticks. ∀_{k,l}⊆Ticks. {i,j} ≠ {k,l} → |i − j| ≠ |k − l|
```

A SONET communication network comprises a number of rings, each joining a number of nodes. A node is installed on a ring using an ADM and there is a capacity bound on the number of ADMs that can be installed on a ring. Each node can be installed on more than one ring. Communication can be routed between a pair of nodes only if both are installed on a common ring. Given the capacity bound and a specification of which pairs of nodes must communicate, allocate a set of nodes to each ring so that the given communication demands are met. The objective is to minimise the number of ADMs used. (This is a common simplification of the full SONET problem, as described in [8])

```
given      nrings : int, nnodes : int, capacity : int
where      nrings ≥ 1, nnodes ≥ 1, capacity ≥ 1
letting    Nodes be int(1..nnodes)
given      demand : set (size m) of set (size 2) of Nodes
find       rings : mset (size nrings) of set (maxsize capacity) of Nodes
minimising ∑_{r∈rings} |r|
such that  ∀_{pair∈demand}. ∃_{r∈rings}. pair ⊆ r
```

In a golf club there are a number of golfers who wish to play together in g groups of size s. Find a schedule of play for w weeks such that no pair of golfers play together more than once. (This transforms into a decision problem and parameterises problem number 10 in CSPLib).

```
1. given      w : int, g : int, s : int
2. where      w > 0, g > 0, s > 0
2. letting    golfers be type of size g × s
3. find       sched : mset(size w) of regpart(size s) of golfers
4. such that  ∀_{week1,week2}⊆sched. ∀_{group1∈week1,group2∈week2}. |group1 ∩ group2| < 2
   Alternative constraint:
4'. such that ∀_{golfer1,golfer2}⊆golfers. (∑_{week∈sched}. together(golfer1, golfer2, week)) < 2
```

Fig. 1. ESSENCE specifications of the Golomb Ruler, SONET, and Social Golfers problems.

must be drawn from the range $0..bound$. Quantified variables must also have finite domains. However, parameters can have infinite domains (e.g. `int`).

ESSENCE Domains correspond to the notion of domains in existing constraint languages. The key advance made by ESSENCE is that it is the first constraint language to support fully compositional domains. For example, a decision variable may have domain `int` $(lb .. ub)$, `set (size s) of int` $(lb .. ub)$, `set (size r) of set (size s) of int` $(lb .. ub)$, etc.

Constraints are built from parameters, constants and decision variables using operators commonly found in mathematics and other constraint specification languages. The language also includes variable binders such as $\forall_x$, $\exists_x$ and Σ_x, where x can range over any specified finite domain (e.g. integer range but not integer). The GRP constraint can be paraphrased "For any two unordered pairs of ticks, $\{i,j\}$ and $\{k,l\}$, if the two pairs are different then the distance between i and j is not the same as the distance between k and l." To clarify, the expression $\{i,j\} \subseteq Ticks$ means that two distinct elements are drawn from $Ticks$ and, without loss of generality, one is called i and the other is called j.

Now consider the specification of the SONET problem (Fig. 1). Notice that *Nodes* is declared to be a domain whose elements are the integers in the range 1..*nnodes*. A subtle point is that the line 3 of the specification is declaring *two* parameters. When parameter *demand* is instantiated to a particular set of sets, the size of the outer set is known. Hence, the value of m is given indirectly. This declaration also requires the inner sets to have cardinality two. The goal is to find a multiset (the rings), each element of which is a set of *Nodes* (the nodes on that ring). The objective is to minimise the sum of the number of nodes installed on each ring. The constraint ensures that any pair of nodes that must communicate are installed on a common ring.

Finally, Fig. 1 gives two specifications of the Social Golfers problem (SGP). The golfers are not referred to individually in the problem description, and so are specified naturally with an unnamed type. The decision variable is represented straightforwardly as a multiset (the fact that it is a set is an implied constraint) of regular partitions, (a `regpart` guarantees equal-sized partitions) each representing a week of play. The specifications differ only in the expression of the socialisation constraint. The constraint at line 4 quantifies over the weeks, ensuring that the size of the intersection between every pair of elements of the corresponding partitions is at most one (otherwise the same two golfers are in a group together more than once). The constraint at line 4′ quantifies over the pairs of golfers, ensuring that they are partitioned together (via the global constraint *together*) over the weeks of the schedule at most once. Note that here we make use of a facility common to constraint languages: treating Booleans as 0/1 for the purpose of counting.

Which of these two alternatives is the more natural is down to the taste of the writer of the specification. This example does, however, highlight that, although ESSENCE allows the user to avoid many more modelling decisions than existing constraint languages, it is possible to write distinct but equivalent ESSENCE specifications. The natural question is whether we are simply requiring users to become experts in writing ESSENCE specifications rather than in traditional constraint programming languages. Central to this question is whether alternative ESSENCE specifications of the same problem are refined to the same models. If so, which of the alternative ESSENCE specifications a user happens to choose is immaterial. However, since our CONJURE refinement system is at a relatively early stage of development [6], we cannot fully answer this question at present.

Let us consider the most pessimistic case: different ESSENCE specifications of the same problem are refined to radically different models. If so, we have replaced one modelling bottleneck with another: expertise in ESSENCE is required to obtain the best models. However, even this is a substantial improvement over the current state of affairs, since the degree of choice is vastly reduced. Typically, an ESSENCE specification can be implemented by many constraint models. Choosing among a small number of ESSENCE specifications is significantly less daunting than choosing among a relatively large number of corresponding constraint models. In future, we aim to reduce the number of ESSENCE specifications of the same problem by employing normalisation procedures to transform each specification into a canonical form, simplifying specification still further.

We close this section by noting that the complexity of the problems expressible in ESSENCE is not clear as, like many other constraint languages, it allows one to build instances with an exponential number of variables (array $[1 \ldots 2^n]$) and exponential domain sizes.

3   The Syntax of ESSENCE

The syntax of well-formed ESSENCE problem specifications is defined by a combination of a grammar and correct type and domain characterisations. The grammar of ESSENCE is detailed in Section 3.1. The domains and the type system are explained in Section 3.2.

3.1   The ESSENCE grammar

The grammar is presented in BNF format using the following conventions: $\{a\}$ stands for a non-empty list of a's, $\{a\}^\circ$ stands for a non-empty list of a's separated by $\circ$'s (where $\circ$ can be any symbol), and $[a]$ stands for nil or one occurrence of a.

Identifiers are strings whose first character is a letter and the rest of the characters are alphanumeric, " _ " or " ' ". Identifiers are keywords introduced by the user, hence they must be different from the built-in reserved keywords (indicated in `teletype` font). A *number* is any string of numeric characters.

Problem specifications in ESSENCE. A grammatically well-formed ESSENCE problem specification is composed of a (possibly empty) *preamble*, a (possibly empty) list of `find` statements, an (optional) *objective* statement and a (possibly empty) list of *constraints*. The preamble includes the `given`, `letting` and `where` statements. Given statements consist of parameter declarations, that is, a list of *domainId*'s, where a *domainId* is an identifier and its domain (e.g. `i:int`). Constant declarations in letting statements associate identifiers to either domains, expressions, or user-defined types. Where statements are conditions that parameters and constants of a valid problem instance must satisfy. For example, in the first specification of Fig. 1 we have the statement `where` $n \geq 0$ that ensures parameter n is non-negative. Find statements are composed by a list of decision variable declarations. Parameter, constant and decision variable declarations comprise the declarations of the specification. The objective statement specifies an objective, either `minimising` or `maximising`, and an arithmetic expression (*arithExpression*) as the objective function of the specification.

$$
\begin{aligned}
spec &::= [\{preamble\}]\ [\{\texttt{find}\ \{domainId\}\text{'}\}]\ [objective]\ [constraints] \\
preamble &::= \texttt{given}\ \{domainId\}\text{'} \\
&\quad \|\ \texttt{letting}\ \{constantId\}\text{'} \\
&\quad \|\ \texttt{where}\ \{expression\}\text{'} \\
objective &::= \{\texttt{minimising}\ arithExpression\} \\
&\quad \|\ \{\texttt{maximising}\ arithExpression\} \\
constraints &::= \{\texttt{such that}\ \{expression\}\text{'}\} \\
constantId &::= identifier\ \texttt{be}\ domain \\
&\quad \|\ identifier\ [\text{``:''}\ domain]\ \texttt{be}\ expression \\
&\quad \|\ identifier\ \texttt{be new enum}\ \{identifier\}\text{'} \\
&\quad \|\ identifier\ \texttt{be new type of size}\ arithExpression \\
domainId &::= identifier\ \text{``:''}\ domain
\end{aligned}
$$

Types and Domains in ESSENCE. The atomic types supported in ESSENCE are: Boolean, integer and user-defined types. The latter are of two kinds, enumeration types and unnamed types. Since user-defined types are associated to an identifier in the constant declarations we use the non-terminals *enumTypeId* and *unnamedTypeId* to represent them in the grammar. The type system of ESSENCE allows the construction of arbitrarily compound (multi) sets, partial and total functions, sequences, permutations, relations, tuples, (regular) partitions and matrices of other types.

Domains are (finite) set of elements all of the same type. One can specify a domain merely by giving its type, in which case the domain consists of all elements of that type (e.g. `i:int`). One can also specify a domain by giving both a type and domain restrictions that select a subset of the the elements of the type (e.g. `i:int(1..3)`). The non-terminals *intRangeRestriction*, *enumRangeRestriction* and *sizeRestriction* represent the domain restrictions that can be applied to integer (e.g. `int(1..3)`), enumeration (e.g. `days(saturday,sunday)` where `days` is an enumeration type consisting of the days of the week) and other types (e.g. `set (size 2) of int`) respectively.

$$
\begin{aligned}
\textit{domain} ::=\ & \text{``(''}\ \textit{domain}\ \text{``)''}\ \ \|\ \texttt{bool} \\
\|\ & \texttt{int}\ [\text{``(''}\ \textit{intRangeRestriction}\ \text{``)''}] \\
\|\ & \textit{enumTypeId}\ [\text{``(''}\ \textit{enumRangeRestriction}\ \text{``)''}] \\
\|\ & \textit{unnamedTypeId} \\
\|\ & \texttt{set}\ [\textit{sizeRestriction}]\ \texttt{of}\ \textit{domain} \\
\|\ & \texttt{mset}\ [\textit{sizeRestriction}]\ \texttt{of}\ \textit{domain} \\
\|\ & \textit{domain}\ \text{``}\mapsto\text{''}\ [\textit{funcClass}]\ \textit{domain} \\
\|\ & \textit{domain}\ \text{``}\rightarrow\text{''}\ [\textit{funcClass}]\ \textit{domain} \\
\|\ & \texttt{rel on}\ [\textit{sizeRestriction}]\ \{\textit{domain}\ [\textit{sizeRestriction}]\}^{\times} \\
\|\ & \texttt{tuple}\ \text{``}\langle\text{''}\ \{\textit{domain}\}\text{'}\ \text{``}\rangle\text{''} \\
\|\ & \texttt{part}\ [\textit{sizeRestriction}]\ \texttt{of}\ \textit{domain} \\
\|\ & \texttt{regpart}\ [\textit{sizeRestriction}]\ \texttt{of}\ \textit{domain} \\
\|\ & \texttt{matrix}\ \text{``[''}\ \texttt{indexed by}\ \{\textit{domain}\}\text{'}\ \text{``]''}\ \texttt{of}\ \textit{domain} \\
\textit{funcClass} ::=\ & \texttt{bij}\ \ \|\ \texttt{inj}\ \ \|\ \texttt{sur}
\end{aligned}
$$

Notice that types are not mentioned explicitly in the grammar, however they exist in ESSENCE as abstractions used by the type checker. Types are produced by removing all the domain restrictions (if any) from domains.

Expressions in ESSENCE. Expressions in ESSENCE can be very complex. For reasons of space we give only a brief description.

The cardinality and negation operators are among the unary operators used to construct expressions. The binary operators can be arithmetic (e.g. +, -), Boolean (e.g. $\land$, $\lor$), comparison (e.g. $=$, $\neq$) or set related (e.g. $\in$, $\cup$). Many global constraints (e.g. `allDiff`, `atmost`, etc), and operators over functions and relations (e.g. `domain`, `inverse`, etc) can be used to compose expressions.

We can name elements of certain types, such as matrix (e.g. `[2, a, b]`), set (e.g. `{2, a, b}`) and multiset (e.g. `#2, a, b#`). An important requirement is that empty (multi)sets must have their domain attached, for example `{}: set of int`. The reason is that the type of a named (multi)set is deduced from its elements and this is not possible for an empty (multi)set.

ESSENCE allows the user to create quantified expressions. Examples of quantified expressions can be found in the constraints of Figure 1. Each quantified expression (e.g. $\forall x, y \in S.x + y \geq z$) consists of a *quantifier* (e.g. $\forall$), a binding expression (e.g. $x, y \in S$), and an expression to quantify (e.g. $x + y \geq z$). The

binding expression implicitly declares the quantified variables (e.g. x, y), and it often contains sub-expression with named elements of various types (e.g. x, y). We show here only some of the cases (*binder*).

$$
\begin{array}{rcl}
quantExp & ::= & quantifier\ bindingExp\ expression \\
quantifier & ::= & \text{``}\forall\text{''},\ \text{``}\exists\text{''},\ \text{``}\varSigma\text{''} \\
bindingExp & ::= & \text{``(''}\ bindingExp\ \text{``)''}\ \|\ domainId\ \|\ binder\ setBindOp\ expression \\
setBindOp & ::= & \text{``}\subset\text{''}\ \|\ \text{``}\subseteq\text{''}\ \|\ \text{``}\in\text{''}\ \|\ \text{``}\notin\text{''} \\
binder & ::= & identifier\ \|\ \text{``\{''}\ \{identifier\}\text{'}\ \text{``\}''} \\
& & \|\ \text{``\#''}\ \{identifier\}\text{'}\ \text{``\#''}\ \|\ \text{``}\langle\text{''}\ \{identifier\}\text{'}\ \text{``}\rangle\text{''}\ \|\ ...
\end{array}
$$

3.2   The Type Checker and the Finite Domain Checker.

The ESSENCE grammar does not give enough information to define a syntactically correct specification. It does not ensure identifiers are combined properly in expressions and that those identifiers have adequate domains. To avoid these problems we need to use the Type Checker (TC) and the Finite Domain Checker (FDC). The TC guarantees all expressions and subexpressions have valid types. The FDC ensures decision and quantified variables have finite domains.

Before performing any of the type and domain verifications we need to construct a table of identifiers from the declarations. In the table every identifier has a unique entry that records the identifier name and its associated information. The information depends on the declaration of the identifier. For parameters, constant domains and decision variables the domain must be saved. Domain and associated expression are stored for the constant expressions whereas we store the type construction for a user-defined type. During the construction of the table we need to check that every identifier is uniquely declared before it is used in another declaration. Also, associated expressions and domains must be type-checked and they must not contain decision variables.

The Type Checker. The TC is described here as a recursive function *typeCheck*: *expression* × *env* ↪ *expression* that, given an expression, returns a well-typed expression with respect to a table of identifiers, also called the "environment" (*env*). *typeCheck* must be applied to all the expressions in a specification. The obtained type of an expression must be consistent with its context, for example, a constraint must be Boolean. When an expression cannot be typed or its type is inconsistent with the contexts it is rejected.

For a built-in constant *c*, *typeCheck* assigns the correct built-in type, except in the cases of empty set and multiset where the type is obtained from the attached domain. For an identifier *i* the function *typeCheck* finds its domain in the environment and returns its type after verifying it is valid. For example, for the domain `matrix indexed by [int(a..b)] of int`, it returns `matrix indexed by [int] of int` after ensuring the indexing type `int` is an ordered type. The function *typeCheck* uses several subfunctions to perform the checking of more complex expressions. One of the most important is *apTypeRule*, the function that applies the type production rules. To clarify the performance of *typeCheck* let us show an example.

Suppose we have the expression `x+4` where `x` has a domain `int(1..4)`. The function *typeCheck* for binary operators (assume □ represents any binary opera-

tor, and a and b stand for any expression) defined as

$$typeCheck\ (a \ \Box \ b)\ env \hookrightarrow apTypeRule\ ((typeCheck\ a\ env) \ \Box \ (typeCheck\ b\ env))$$

obtains the integer type for `a` and `4` after type-checking them. Then it applies the type rule for integer addition

$$a{:}\texttt{int} + b{:}\texttt{int} \rightsquigarrow (a+b){:}\texttt{int}$$

and returns `(x+4):int`.

That is, the expression `(x+4)` has type integer. *typeCheck* performs similarly for other operators, types and type rules. For quantified expressions, *typeCheck* also includes the quantified variables in the current environment.

The Finite Domain Checker. Decision variables and quantified variables (in a domain binding expression) must have a finite domain. Parameters and constants do not need a finite domain since they will have a finite value assigned when implementing an instance. We define below the rules to identify a finite domain (FDOMAIN).

$$
\begin{aligned}
\text{FDOMAIN} ::=\ &\texttt{bool}\ \parallel\ \texttt{int}(iRange)\ \parallel\ enumIdentifier[(eRange)] \\
\parallel\ &\mathit{unnamedIdentifier} \\
\parallel\ &\texttt{set}\ [sizeRestrict]\ \texttt{of}\ \text{FDOMAIN} \\
\parallel\ &\texttt{mset}\ [sizeRestrict]\ \texttt{of}\ \text{FDOMAIN} \\
\parallel\ &\text{FDOMAIN}\ \text{``}->\text{''}\ [funcType]\ \text{FDOMAIN} \\
\parallel\ &\text{FDOMAIN}\ \text{``}|->\text{''}\ [funcType]\ \text{FDOMAIN} \\
\parallel\ &\texttt{tuple}\ \text{``}\langle\text{''}\ \{\text{FDOMAIN}\}\text{'}\ \text{``}\rangle\text{''} \\
\parallel\ &\texttt{rel}\ [sizeRestrict]\ \{\text{FDOMAIN}\ [sizeRestrict]\}^{\times} \\
\parallel\ &\texttt{part}\ [sizeRestrict]\ \texttt{of}\ \text{FDOMAIN} \\
\parallel\ &\texttt{regpart}\ [sizeRestrict]\ \texttt{of}\ \text{FDOMAIN}
\end{aligned}
$$

4 The Semantics of ESSENCE

This section presents a semantic account of ESSENCE that identifies the conditions under which an ESSENCE specification is satisfied—that is, what assignments to decision variables satisfy what specifications. To be clear, our focus is defining the truth conditions of the language, not on defining the behaviour of a decision procedure for satisfiability or any other program. Our semantic account specifies a denotation function from ESSENCE specifications and assignments to the truth values. This function is not total; the denotation of a specification relative to an assignment may be undefined. The denotation function is defined compositionally, and thus applies to every expression and subexpression of ESSENCE.

The semantic rules assign denotations to ESSENCE specifications in which each expression is annotated with its type. We also assume that specifications are normalised in that each statement that declares multiple symbols has been replaced with a sequence of statements each declaring a single symbol. Similarly, a `such that` statement with multiple constraints is assumed to have been replaced with multiple `such that` statements, each with a single constraint. The semantic rules do not deal with `given` statements. It suffices to consider a statement of the form "`given` p" to be synonymous with a statement of the form "`letting` p `be` exp", where exp is an expression provided by the user to define the instance.

The denotation of an expression P is taken relative to three assignments: an assignment A of values to decision variables, an assignment θ of values to user-defined identifiers, and an assignment g of values to quantified variables. The denotation of P relative to A, g and θ is written as $[\![P]\!]^{A,\theta,g}$. Thus, $[\![\,\cdot\,]\!]$ is a partial function of arity four. A consequence of the definition of the denotation function is that if P is an ESSENCE specification and A is an assignment to its decision variables, then the value of $[\![P]\!]^{A,\theta,g}$ is the same regardless of θ and g. Thus, we can simply write $[\![P]\!]^{A}$ and speak of the denotation of P with respect to A.

The denotation function is not total; $[\![exp]\!]^{A,\theta,g}$ may be undefined. This is necessary to handle cases such as where an array index is out of bounds or a divisor is zero. An attempt to do otherwise leads to problems. If "$5/0 = 1$" is false, then we must accept that "$5/0 \neq 1$" is true. What, then, does one say about the truth of "$5/0 = 4/0$"? We also say that the denotation of an ESSENCE specification is undefined if any of its `where` constraints are not met or if a decision variable is assigned a value outside its domain. The definition of the denotation function should not be considered as an incomplete definition; rather it is a complete definition of a partial function.

Our primary intuition regarding partiality is that a specification of a decision problem is satisfied by an assignment if the denotation of the specification with respect to the assignment is T; and this is the case even if the denotation is undefined with respect to other assignments. Though constraint programming languages typically do not have semantic specifications, their behaviour often conflicts with our intuition; for example, if one assignment leads to an array index being of bounds an exception is raised and (by default) the program is halted, even if other computation paths lead to a solution.

In the rules that follow, an expression of the form $s; S$ stands for a sequence of ESSENCE statements, the first of which is s and the remainder of which is the possibly empty sequence S. If σ is an assignment of any kind (either A, θ, or g), then $\sigma[x \mapsto d]$ is the assignment that is identical to σ with the possible exception that it maps x to d.

Let us first handle `letting` and `where` statements. A `letting` statement declares a new identifier and assigns it a value; the semantics accounts for this assignment by recording it in θ.

- $[\![\texttt{letting } c \texttt{ be } exp; R]\!]^{A,\theta,g} = [\![R]\!]^{A,\theta[c \mapsto e],g}$, where $e = [\![exp]\!]^{A,\theta,g}$.
- $[\![\texttt{letting } e \texttt{ be new enum } c_1, \ldots, c_n; R]\!]^{A,\theta,g} = [\![R]\!]^{A,\theta',g}$, where $\theta' = \theta[e \mapsto \{c'_1, \ldots, c'_n\}, c_1 \mapsto c'_1, \ldots, c_n \mapsto c'_n]$, and $\{c'_1, \ldots, c'_n\}$ is any arbitrary set of size n. This set is totally ordered by "$\leq$": $c'_1 \leq c'_2 \leq \cdots \leq c'_n$.
- $[\![\texttt{letting } u \texttt{ be new type of size } exp); R]\!]^{A,\theta,g} = [\![R]\!]^{A,\theta[u \mapsto U],g}$, where U is an arbitrary set of size $[\![exp]\!]^{A,\theta,g}$.
- $[\![\texttt{where } constraint; R]\!]^{A,\theta,g} = [\![R]\!]^{A,\theta,g}$ provided $[\![constraint]\!]^{A,\theta,g} = T$.

The last semantic rule does not specify a denotation if the constraint is not satisfied. Our convention is that where a denotation is not specified, it is undefined.

The only role of `find` statements that must be accounted for by the semantics is to ensure that the denotation of a specification relative to an assignment A is undefined if A maps a decision variable to a value not in its domain. A `minimising` or `maximising` statement must be handled together with all the `find` statements as optimisation is over all assignments to the decision variables. Thus, we give

two rules: the first handles all the `find` statements in the absence of optimisation and the second handles minimisation by considering all solutions to the `find` and `such that` statements. The rule for maximisation is similar to that for minimisation and therefore is not shown.

- $[\![\texttt{find } x_1{:}\tau_1; \ \ldots; \ \texttt{find } x_n{:}\tau_n; \ R]\!]^{A,\theta,g} = [\![R]\!]^{A,\theta,g}$ provided $A(x_i) \in [\![\tau_i]\!]^{A,\theta,g}$ $(1 \leq i \leq n)$ and R contains no `find`, `minimising` or `maximising` statements.
- $[\![\texttt{find } x_1{:}\tau_1; \ldots; \texttt{find } x_n{:}\tau_n; \ \texttt{minimising } exp; \ R]\!]^{A,\theta,g}$
  $= T$ if $[\![FR]\!]^{A,\theta,g} = T$ and for all A' such that $[\![FR]\!]^{A',\theta,g} = T$, $[\![exp]\!]^{A,\theta,g} \leq [\![exp]\!]^{A',\theta,g}$,
  $= F$ if $[\![FR]\!]^{A,\theta,g} = F$,
  $= F$ if $[\![FR]\!]^{A,\theta,g} = T$ and for some A' such that $[\![FR]\!]^{A',\theta,g} = T$, $[\![exp]\!]^{A,\theta,g} > [\![exp]\!]^{A',\theta,g}$,
  where FR is $(\texttt{find } x_1{:}\tau_1; \ldots; \texttt{find } x_n{:}\tau_n; \ R)$.

As the first of the two rules above does not specify a denotation to a find statement that does not meet the stated provision, by our convention, the denotation of such a statement is undefined.

The only role that domains play in the semantics is in the previous equation, which involves only finite domains. The denotations of some finite domains are defined as follows—the remainder are similar to these. Here we consider τ_o to be an ordered type.

- $[\![\texttt{bool}]\!]^{A,\theta,g} = \{T, F\}$
- $[\![\tau_o \ (r_1, \ldots, r_n)]\!]^{A,\theta,g} = [\![\tau_o \ (r_1)]\!]^{A,\theta,g} \cup [\![\tau_o \ (r_2, \ldots, r_n)]\!]^{A,\theta,g}$, provided $n \geq 2$.
- $[\![\tau_o \ (l..u)]\!]^{A,\theta,g} = \{i \mid [\![l]\!]^{A,\theta,g} \leq i \leq [\![u]\!]^{A,\theta,g}\}$.
- $[\![\tau_o \ (exp)]\!]^{A,\theta,g} = \{[\![exp]\!]^{A,\theta,g}\}$, where exp is not of the form $l..u$ or $r_1, \ldots, r_n$.
- $[\![\texttt{set (size } exp) \texttt{ of } \tau]\!]^{A,\theta,g} = \{S \subseteq [\![\tau]\!] \mid [\![exp]\!]^{A,\theta,g} = |S|\}$.
- $[\![\texttt{set (maxsize } exp) \texttt{ of } \tau]\!]^{A,\theta,g} = \{S \subseteq [\![\tau]\!] \mid [\![exp]\!]^{A,\theta,g} \geq |S|\}$.
- $[\![\texttt{matrix indexed by } [d_1, \ldots, d_n] \texttt{ of } \tau]\!]^{A,\theta,g} = [\![d_1]\!]^{A,\theta,g} \times \cdots \times [\![d_n]\!]^{A,\theta,g} \to [\![\tau]\!]^{A,\theta,g}$.

Notice that a matrix denotes a function that is total when the indices are within bounds, and undefined when they are out of bounds.

A sequence of `such that` statements is satisfied if the first statement is satisfied and the rest are satisfied. The empty sequence is always satisfied.

- $[\![\texttt{such that } C; R]\!]^{A,\theta,g} = T$ if $[\![C]\!]^{A,\theta,g} = [\![exp_2]\!]^{A,\theta,g} = T$
  $= F$ if $[\![C]\!]^{A,\theta,g} = F$
  $= F$ if $[\![R]\!]^{A,\theta,g} = F$.
- $[\![\ ]\!]^{A,\theta,g} = T$.

Now consider the atomic expressions. In all assignments, "T", "F", "1", "2", "3", etc. denote T, F, 1, 2, 3, etc. For other atomic expressions we have:

- $[\![c]\!]^{A,\theta,g} = \theta(c)$ where c is any user-defined identifier.
- $[\![x]\!]^{A,\theta,g} = A(x)$ where x is a decision variable.
- $[\![x]\!]^{A,\theta,g} = g(x)$ where x is a quantified variable.

Now consider the operators that are used to build up constraints and other expressions. We mostly focus here on the treatment of undefined denotations as in other respects the semantics is both obvious and similar to that of other languages. Our intuition is that existentially-quantified variables should behave like decision variables, which suggests the following definition:

- $[\![\exists_{x:\tau}.constr]\!]^{A,\theta,g} = T$ if $[\![constr]\!]^{A,\theta,g[x \mapsto d]} = T$ for some $d \in [\![\tau]\!]^{A,\theta,g}$
  $= F$ if $[\![constr]\!]^{A,\theta,g[x \mapsto d]} = F$ for all $d \in [\![\tau]\!]^{A,\theta,g}$.

Another intuition is that disjunction should behave like existential quantification, suggesting the following rule:

- $[\![exp_1 \vee exp_2]\!]^{A,\theta,g} = F$ if $[\![exp_1]\!]^{A,\theta,g} = [\![exp_2]\!]^{A,\theta,g} = F$
  $= T$ if $[\![exp_1]\!]^{A,\theta,g} = T$
  $= T$ if $[\![exp_2]\!]^{A,\theta,g} = T$.

The rules for conjunction and universal quantification are readily obtained from the above definitions by interchanging "T" and "F."

Now consider the division operator:

- $\llbracket exp_1/exp_2 \rrbracket^{A,\theta,g} = \llbracket exp_1 \rrbracket^{A,\theta,g} / \llbracket exp_2 \rrbracket^{A,\theta,g}$

Observe that since division by zero is not defined, if exp_2 denotes zero, then the denotation of exp_1/exp_2 is undefined.

Some operator symbols are overloaded; the operator they denote depends on the type (never the denotation) of their operands. Consider $\cup$, which can denote set union or multiset union. This operator requires two syntactic rules, and it requires a semantic rule for each.

- $\llbracket exp_1 \cup exp_2 \rrbracket^{A,\theta,g} =$ the multiset union of $\llbracket exp_1 \rrbracket^{A,\theta,g}$ and $\llbracket exp_2 \rrbracket^{A,\theta,g}$, provided both exp_1 and exp_2 are of type $\mathtt{mset\ of\ } \tau$.
- $\llbracket exp_1 \cup exp_2 \rrbracket^{A,\theta,g} =$ the set union of $\llbracket exp_1 \rrbracket^{A,\theta,g}$ and $\llbracket exp_2 \rrbracket^{A,\theta,g}$, provided both exp_1 and exp_2 are of type $\mathtt{set\ of\ } \tau$.

The rule for indexing into a matrix is straightforward.

- $\llbracket M[i_1,\ldots,i_n] \rrbracket^{A,\theta,g} = \llbracket M \rrbracket(\llbracket i_1 \rrbracket^{A,\theta,g},\ldots,\llbracket i_n \rrbracket^{A,\theta,g})$.

5   An Evaluation of the Use of ESSENCE

To evaluate the effectiveness of ESSENCE we decided to specify a large suite of problems in the language and reflect on the process and resulting specifications. A suite of 57 problems was selected, 25 drawn from `www.csplib.org`, and 32 from constraint journals and conferences. This lists includes both problems which are theoretical in nature and problems drawn from the real world.

The catalogue[2] contains ESSENCE specifications for all the problems as well as some previously-published specifications in other languages, including Z, ESRA, OPL and $\mathcal{F}$. The catalogue demonstrates the expressiveness and elegance of ESSENCE when compared to other constraint languages. In all cases the length of the specification was in proportion to the size of the problem statement, with large examples being just as easy to read.

The task of specifying the problem suite in ESSENCE was undertaken by the second author after studying computer science for three years as an undergraduate. He had no previous experience of constraint programming or logic programming. At the outset the language had no formal description so he had to learn the language from a small set of example specifications. He found that he was able to develop his understanding of ESSENCE by drawing heavily on his understanding of the conventional set and function operators of discrete mathematics.

The process of writing an Essence specification began by obtaining an unambiguous natural language description of the problem. The flexibility of the language allowed specifications to be written directly from this description, starting with the parameters, and following with the decision variable, objective, and constraints.

Attention had to be paid to abstraction during specification in order to make full use of the language, as some of the problems were described in the literature in terms of low-level objects such as arrays. With the goal of producing the most

---

[2] available at `www.cs.york.ac.uk/aig/constraints/AutoModel/`

abstract specification possible there was usually a single choice for the type of the decision variable. In the SONET problem the configuration can be represented as a set of sets of nodes or as a relation from rings to nodes. Both possibilities are present in the problem catalogue.

Once several specifications had been written, the process became easier as the specifications contained common idioms that could be reused. Patterns were present in the high level specifications that would not necessarily appear in the concrete constraint programs due to differing representations of complex objects. Certain constructs were repeated in their entirety, for example minimising a total cost by summing the costs of each item in a set.

6   Comparison of ESSENCE with other languages

Algebraic modelling languages, dating from the 1970s and originating in the field of mathematical programming (e.g. GAMS [9] and MGG [10]), made three significant advances. Firstly they were developed to simplify the user's role in solving mathematical programming problems, providing a syntax much closer to the expression of these problems found in the literature. Secondly, they are declarative, characterising the solutions to a problem rather than how the solutions are to be found. This lifts a burden from the user, and allows different solvers to be easily applied to the same problem. Finally, they separate the specification from instance data, allowing parameterised specifications of problems. As mathematical programming solvers become increasingly powerful and complex, the importance of these languages has increased (e.g. AMPL [11]).

The success of algebraic modelling languages as an interface to mathematical programming solvers suggests that a similar approach might be fruitful for constraint solving. From early in the development of the field, the ALICE language [12] shares many features with algebraic modelling languages of the time, including its declarative nature. Following this, however, the trend in constraint solving was to extend a general-purpose programming language, such as Prolog (ECL^{i}PS^e [13], CHIP [14]) or C++ (ILOG Solver [15]) with a constraint library. Although this approach is undoubtedly powerful and efficient, it is not conducive to reducing the modelling bottleneck as such systems are often complex, requiring knowledge of the host general-purpose language, and interleave the specification of a problem with how it is solved.

More recently, in parallel with an increasing awareness of the modelling bottleneck in constraints, abstract constraint modelling has re-emerged in languages such as EaCL [16], OPL [17], ESRA [4], and $\mathcal{F}$ [3]. OPL has enjoyed particular success, not least because of its ability to support models intended to be solved by a combination of mathematical and constraint programming techniques. ESRA and $\mathcal{F}$ are more abstract than OPL, providing support for relations and functions respectively.

However, to be truly valuable, a modelling language should not force the user to make modelling decisions when a specification is written down. For example, if the problem involves partitioning, it should not be necessary for the user to decide to represent the partition as, say, a constrained set of sets. Otherwise, the onus is

on the user to make the most effective choice. If an ineffective choice is selected, the early commitment to a representation may inhibit automated attempts to improve the resultant model.

Therefore, a modelling language should have a level of abstraction above that at which modelling decisions are made. Current algebraic/constraint modelling languages are not abstract enough. It may be argued that general specification languages, such as B, Z, or (more focused on combinatorial problems) NP-Spec are sufficiently abstract to avoid forcing the user to make modelling decisions. However, we believe that an abstract specification language tailored to constraint problems provides the most natural means of specifying such problems.

In order to illustrate the advances made by Essence, we compare the specifications of the SGP in Essence, NP-Spec, esra and opl. Both an English definition and an Essence specification of this problem are given in Fig. 2, specifications in esra (from [4]), opl (written by the authors) and NP-Spec[3] are given in Fig. 6. Each specification is divided into 4 sections. The first declares any parameters and types, the second declares the variables, the third imposes that the variables represent a multiset of partitions and the fourth that the "sociability constraint" (that is no pair of players are in the same group more than once) is satisfied.

The most natural way in which to represent the solution of this problem is as a multiset (as the weeks are not distinguished) of partitions of the players, where each partition gives the games played in a week. The Essence specification is able to capture this entirely in the decision variable it uses. While NP-Spec provides a partition type, it is not possible to create either a list or multiset of partitions and so it cannot be used. Each of the NP-Spec, opl and esra specifications therefore uses a specific implementation of multisets of partitions for the variables, requiring the user to make modelling decisions. The NP-Spec and esra specifications both implement the multiset of partitions by introducing a range of values which represent which group a player plays in, and then use a function from "players × weeks" to the group that player is in during that week. The opl specification's decision variable is a refinement of this, where the function is represented as a 2 dimensional matrix indexed by players and weeks, with elements drawn from a range to denote the group.

Each of the esra, opl and NP-Spec specifications have to give a constraint which imposes the decision variables represent a multiset of regular partitions. These constraints are contained in the 3rd section of each specification.

As they label the players, weeks and groups, the esra, opl and NP-Spec specifications introduce a large amount of symmetry which is present in the original specification. The symmetry of the weeks and players can be removed by using unnamed types, but the symmetry of the groups is more subtle. Labelling groups in different weeks using the same type introduces a dependancy between the groups in each different weeks not present in the specification. The true symmetry group allows the groups to be freely permuted independently in each week. By not having to label the groups, the ESSENCE specification avoids this problem. The full extent of the symmetries of the groups has been missed previously

---

[3] from http://www.dis.uniroma1.it/~cadoli/research/projects/NP-SPEC/

(for example [18]), which shows how the use of Conjure can aid even expert modellers.

One unusual feature in esra's definition of "Sched", on line 2, is the notation $\to^{groupsize*weeks}$. This represents that each element of Group is mapped to by exactly *groupsize * weeks* assignments. This is implied by the fact that this function is representing a multiset of partitions, but is insufficient to fully impose this condition, as this requires the stronger condition that each group contains *groupsize* players in each week.

The main constraint of the SGP, contained in the denoted as part four of each specification, is that no pair of players may play together more than once. Both the opl and esra specifications impose this constraint by quantifying over the players. NP-Spec's specification implements this constraint by imposing that there must not exist two distinct players and two distinct weeks such that the players play together in both weeks. This can be seen as an alternative method of specifying the constraint in the Essence specification, using the common transformation that instead of imposing less than n elements of a set satisfy a condition, it is equivalent to check that there is no subset of size n where all elements satisfy the condition.

ESRA

1a	cst weeks, groups, groupsize : N
1b	dom $Players = 1..groups * groupsize$, $Weeks = 1..weeks$, $Groups = 1..groups$;
2	var $Sched : (Players \times Weeks) \to^{groupsize \times weeks} Groups$
	solve
3	$\forall(h : Groups, w : Weeks)count(groupsize)(p : Players|Sched(p,v) = h)$
4	$\wedge\forall(p1 < p2 : Players)count(0...1)(w : Weeks|Sched(p1,v) = Sched(p2,v))$

OPL

1a	int weeks = ... ; int groups = ... ; int groupsize = ... ;
1b	range Weeks 1..weeks; range Groups 1..groups; range Players 1..groups*groupsize;
2	var Groups Schedule[Players, Weeks];
	subject to {
3a	forall(w in Weeks & g in Groups) (sum(p in Players) (Schedule[p,w] = g) = groupsize);
4a	forall(ordered p1,p2 in Players) (sum(w in Weeks) (Schedule[p1,w] = Schedule[p2,w]) < 2);
	}

NP-Spec

	DATABASE
1	weeks = 6; groups = 8; groupsize = 4;
	SPECIFICATION
2	IntFunc({1..groups*groupsize}><{1..weeks}, Schedule, 1..groups).
3	fail <-- COUNT(Schedule(*,W,Gr),X), X != groupsize.
4	fail <-- Schedule(P1,W1,Gr1), Schedule(P2,W1,Gr1), P1 != P2,
	Schedule(P1,W2,Gr2), Schedule(P2,W2,Gr2), W1 != W2.

Fig. 2. esra,opl and NP-Spec specifications of the Social Golfers Problem

7 Conclusion

Essence allows combinatorial problems to be specified at a high level of abstraction. The result is that problems can be specified without (or almost without) modelling them. The central, unique feature of Essence is that it supports complex, arbitrarily-nested types. Consequently, a problem that requires finding a

complex combinatorial object can be directly specified by using a decision variable whose type is precisely that combinatorial object.

References

1. Garey, M., Johnson, D.: Computers and Intractability. W. H. Freeman (1979)
2. Gervet, C.: Conjunto: Constraint logic programming with finite set domains. In Bruynooghe, M., ed.: Logic Programming - Proceedings of the 1994 International Symposium, Massachusetts Institute of Technology, The MIT Press (1994) 339–358
3. Hnich, B.: Function Variables for Constraint Programming. PhD thesis, Computer Science Division, Department of Information Science, Uppsala University (2003)
4. Flener, P., Pearson, J., Agren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Proceedings of LOPSTR '03: Revised Selected Papers. Volume 3018 of LNCS. (2004)
5. Cadoli, M., Ianni, G., Palopoli, L., Schaerf, A., Vasile, D.: NP-SPEC: An executable specification language for solving all problems in NP. Computer Languages **26** (2000) 165–195
6. Frisch, A.M., Jefferson, C., Martínez Hernández, B., Miguel, I.: The rules of constraint modelling. In: Proceedings of the 19th International Joint Conferences on Artifical Intelligence. (2005) 109 – 116
7. Gent, I.P., Walsh, T.: CSPLib: A problem library for constraints. `www.csplib.org` (2005)
8. Frisch, A.M., Hnich, B., Miguel, I., Smith, B.M., Walsh, T.: Transforming and refining abstract constraint specifications. In: Proceedings of the 6th Symposium on Abstraction, Reformulation and Approximation. Volume 3607 of LNCS., Springer (2005) 76–91
9. Brooke, A., Kendrick, D., Meeraus, A.: GAMS: A Users' Guide. The Scientific Press, Danvers, Massachusetts (1988)
10. Simons, R.: Mathematical programming modeling using MGG. IMA Journal of Mathematics in Management **1** (1987) 267–276
11. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Second edn. Thomson/Brooks/Cole, Pacific Grove, California (2003)
12. Lauriere, J.L.: ALICE: A language and a program for stating and solving combinatorial problems. Artificial Intelligence **10** (1978) 29–127
13. Cheadle, A., Harvery, W., Sadler, A.J., Schimpf, J., Shen, K., Wallace, M.: ECLiPSe: An introduction. Technical Report IC-Parc-03-1, Imperial College London (2003)
14. Aggoun, A., Beldiceanu, N.: Overview of the CHIP compiler system. In Benhamou, F., Colmerauer, A., eds.: Constraint Logic Programming: Selected Research. MIT Press, London (1993) 421–436
15. ILOG: ILOG Solver 5.1 User's Manual. (2000)
16. Mills, P., Tsang, E., Williams, R., Ford, J., Borrett, J.: EaCL 1.5: An easy abstract constraint optimisation programming language. Technical report, University of Essex, Colchester, UK (1999)
17. Van Hentenryck, P.: The OPL Optimization Programming Language. The MIT Press (1999)
18. Flener, P., Frisch, A., Hnich, B., Kızıltan, Z., Miguel, I., Walsh, T.: Matrix modelling. In: Proceedings of the CP'01 Workshop on Modelling and Problem Formulation. (2001) 1–7