ESSENCE: A Constraint Language for Specifying Combinatorial Problems

Alan M. Frisch[1] **Warwick Harvey**[2] **Chris Jefferson**[3]
Bernadette Martínez-Hernández[1] **Ian Miguel**[2]

[1]Artificial Intelligence Group, Dept. of Computer Science, Univ. of York, UK. {frisch, berna}@cs.york.ac.uk

[2]School of Computer Science, Univ. of St. Andrews, UK. {wh, ianm}@cs.st-and.ac.uk

[3]Oxford University Computing Laboratory, Univ. of Oxford, UK. chrisj@comlab.ox.ac.uk

Abstract. ESSENCE is a formal language for specifying combinatorial problems, in a manner similar to natural rigorous specifications that use a mixture of natural language and discrete mathematics. ESSENCE provides a high level of abstraction, much of which is the consequence of 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, providing for example sets of partitions, sets of sets of partitions, and so forth. Therefore, a problem that requires finding a complex combinatorial object can be specified directly by using a decision variable whose type is precisely that combinatorial object.

1 Introduction

This paper describes ESSENCE, a 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 abstract problem specifications that are similar to rigorous specifications that use a mixture of natural language and discrete mathematics, such as those catalogued by Garey and Johnson [14].

ESSENCE is intended to be accessible to anyone with a background in discrete mathematics; no expertise in constraint programming should be needed. Our working hypothesis has been that this could be achieved by modelling the language after the rigorous specifications that are naturally used to describe combinatorial problems rather than developing some form of logical language, such as Z [32] or NP-SPEC [3]. This has resulted in a language that to a first approximation is a constraint language, such as OPL [33], $\mathcal{F}$ [18] or ESRA [6],

enhanced with features that greatly increase its level of abstraction. Most importantly, as a combinatorial problem requires finding one or more combinatorial objects, Essence provides decision variables whose domain elements are combinatorial objects, and constraints that operate on them. This enables problems to be stated directly and naturally; without decision variables of the appropriate type a user would have to "model" the problem by encoding the desired combinatorial object as a collection of constrained decision variables of a simpler type, such as integers.

As our motivations — and hence our methodology and results — for developing Essence differ greatly from those that have led to the development of other constraint languages, it is important to consider these carefully at the outset.

Our primary motivation comes from our ongoing study of the automation of finite-domain constraint modelling, the process of reducing a given problem to an equivalent finite-domain constraint satisfaction or optimisation problem (CSP) in which all the domains and constraints are supported by the intended solver technology. In current practice, this process is conducted manually and unsystematically — that is, it is still an art. As current solvers provide decision variables whose domains contain atomic values or, sometimes, finite sets of atomic values, models must always be in terms of these. As an example, consider the Social Golfers Problem (SGP), which requires partitioning a set of golfers into groups of a given size in each week of a tournament, subject to a certain constraint. Thus, the goal is to find a set of partitions that satisfies the constraint. A model for the SGP must therefore represent this set of partitions by a collection of constrained variables whose domains are either atoms or finite sets of atoms. There are at least 54 ways that this can be done [11].

In order to automatically generate models for a problem, one must start with a formal specification of the problem that is sufficiently abstract that no modelling decisions have been made in constructing it. Thus, the formal language for writing specifications must provide a level of abstraction above that at which modelling decisions are made. We refer to such a language as a *problem specification language* and distinguish it from modelling languages, whose purpose is to enable the specification of models. Thus, designing a problem specification language is a *prerequisite* to studying automated modelling and the goal of this paper is to put forward a language that makes a large step towards satisfying this challenging prerequisite.

Another motivation for designing Essence is that formal problem specifications could facilitate communication between humans better than the informal specifications that are currently used, and further benefits could accrue from standardising a problem specification language. For example, the informal problem descriptions in the CSPLib problem library (http://csplib.org) could be replaced or supplemented by formal ones, but doing so requires the availability of a *problem* specification language, as the founders of CSPLib wisely insisted on cataloguing problems rather than models. Using formal specifications for human communication imposes the requirement that the language is *natural* — that is, it is similar to the manner in which people think of problems

and the style in which they specify them informally. Naturalness is also an important property for the input language of an automated modelling system; one cannot claim to have an automated modelling system if using it requires a major translation of the problem into the system's input language. Finally, expertise in constraint modelling or constraint solving should not be needed for writing a formal problem specification, be it for human communication or for input to an automated modelling system.

Three primary objectives have driven the design of Essence. This section has discussed two: The language should be natural enough to be understood by someone with a background in discrete mathematics and it should provide a high level of abstraction. These two objectives are related; providing an appropriate level of abstraction is necessary to achieve naturalness. The third design objective is that problems specified in the language can be effectively mapped to CSPs. As an illustration, one consequence of this objective is that every decision variable in Essence is associated with a finite domain.

The remainder of this paper introduces Essence and argues that it largely meets its design objectives. A rigorous grammar for the current version of Essence, along with other related documents, can be found at our automated constraint modelling website[1]. The reader can use this website best as a reference to provide details not provided by the broad, explanatory overview provided by the paper. A formal semantics for an earlier version of Essence can be found in [8].

Essence is evolving rapidly. Version 1.2.0 is used within this paper; a previous paper [9] was based on version 1.1.0. While the central philosophy of the language, to provide a natural, expressive constraint specification language — whose level of abstraction is above that at which modelling decisions are made — remains, there have been significant changes from version 1.1.0 to version 1.2.0. The type system has been carefully revised and updated to aid expressivity, and a number of grammatical changes have improved perspicuity and resolved ambiguities.

The paper concludes with a frank assessment of the shortcomings of the current language and our plans to remedy these shortcomings. To take a broader view, one might question whether an endeavour such as Essence can really remove the constraint modelling "bottleneck", or whether it simply pushes the problem to a higher level of abstraction — after all, it is a simple matter to write alternative Essence specifications of the same problem, as we will see. Is expertise required to select the "best" one? The point is that Essence has the expressivity to enable users to specify a problem in a way that is natural to them. Although one user might write a quite different specification from another, because an Essence specification is abstract there remains significant scope for making good constraint modelling decisions, irrespective of the specification written.

[1] http://www.cs.york.ac.uk/aig/constraints/AutoModel/

2 An Introduction to Essence by Example

Through the presentation of examples, this section both introduces Essence and demonstrates its naturalness.

We begin by asking the reader to examine the first Essence specification given in Fig. 1. This is a specification of a well-known problem. Can you identify it?

Many readers are able to identify this as the Knapsack decision problem because the Essence specification is nearly identical to the specification given by Garey and Johnson [14, problem MP9, page 247]:

> INSTANCE: Finite set U, for each $u \in U$: a size $s(u) \in \mathbb{Z}^+$, a value $v(u) \in \mathbb{Z}^+$ and positive integers B and K.
>
> QUESTION: Is there a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$?

The similarity of this Essence specification to the naturally-arising problem specification illustrates our main point: problems are often specified rigorously via discrete mathematics. Essence is based on the notation and concepts of discrete mathematics. Hence, someone able to understand rigorous problem specifications that employ discrete mathematics can, with little training, also understand Essence specifications. This is a significant advantage since far more people are familiar with discrete mathematics than constraint programming.

Natural problem specifications, such as the one above, identify what is given (the parameters of the problem), what combinatorial objects must be found (the decision variables) and what constraints the objects must satisfy to be a solution. The specification might also introduce some terminology, give an objective function if the problem is an optimisation problem, and identify conditions that must be met by the parameter values. Essence supports these components of a problem specification with seven kinds of statements, signalled by the keywords `given`, `where`, `letting`, `find`, `maximising`, `minimising` and `such that`. `letting` statements declare identifiers and user-defined types. `given` statements declare parameters, whose values are input to specify the instance of the problem class. Parameter values are not part of the problem specification; they define individual instances of the problem and are provided in a separate input specification. `where` statements dictate allowed parameter values; only allowed values designate valid problem instances. `find` statements declare decision variables. A `minimising` or `maximising` statement gives the objective function, if any. Finally, `such that` statements give the problem constraints.

Essence has an eighth kind of statement, signalled by the keyword `language`, which indicates that the specification is an Essence specification and gives the version number of the language. As Essence is a rapidly evolving language, it is useful to provide a formal way to record the version number; this can be checked for compatibility during processing.

Now consider the specification of the Golomb Ruler Problem (GRP) in Fig. 1, and the instance obtained by letting n be 4. The domain of the decision

Mystery Problem:

```
language    Essence 1.2.0
given       U new type enum,
            s, v : function (total) U ⟶ int (1..),
            B, K: int (1..)
find        U': set of U
such that   ∑ u ∈ U'.s(u) ≤ B,
            ∑ u ∈ U'.v(u) ≥ K
```

Golomb Ruler Problem (GRP): Given n, put n integer ticks on a ruler of size m so that all inter-tick distances are unique. Minimise m. (This is problem 6 in CSPLib.)

```
language    Essence 1.2.0
given       n : int          $ this could be int (1..)
where       n ≥ 1
letting     bound be 2**n
find        Ticks : set (size n) of int (0..bound)
minimising  max(Ticks)
such that   ∀pair1, pair2 : set (size 2) of int ⊆ Ticks . pair1 ≠ pair2 →
              max(pair1) − min(pair1) ≠ max(pair2) − min(pair2)
```

SONET Problem: A SONET communication network comprises a number of rings, each joining a number of nodes. A node is installed on a ring using an add-drop multiplexer (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 by Frisch *et al.* [10])

```
language    Essence 1.2.0
given       nrings, nnodes, capacity : int (1..)
letting     Nodes be domain int (1..nnodes)
given       demand : set of set (size 2) of Nodes
find        network : mset (size nrings) of set (maxSize capacity) of Nodes
minimising  ∑ ring ∈ network . |ring|
such that   ∀pair ∈ demand . ∃ring ∈ network . pair ⊆ ring
```

Social Golfers Problem (SGP): 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.)

```
language    Essence 1.2.0
given       w, g, s : int (1..)
letting     golfers be new type of size g * s
find        sched : set (size w) of partition (numParts g, partSize s) from golfers
such that   ∀week1,week2 ∈ sched . week1 ≠ week2 →
                ∀group1 ∈ parts(week1), group2 ∈ parts(week2) . |group1 ∩ group2| < 2
Alternative constraint:
such that   ∀golfer1,golfer2 : golfers . golfer1 ≠ golfer2 →
                (∑ week ∈ sched . toInt(together(golfer1, golfer2, week)) < 2)
```

Fig. 1. Essence specifications of four problems.

variable *Ticks* is instantiated to be `set {size 4} of int {0..16}`, which denotes the domain containing all sets of four elements drawn from $\{0, .., 16\}$. The reason we use special brackets is explained later, but is irrelevant for the informal reading of specifications. Consider an assignment mapping *Ticks* to $\{0, 1, 3, 7\}$. Is this a solution to this instance? If we have succeeded in our goal of making Essence specifications natural to those with a discrete mathematics background, then it should be clear that it is: given any two distinct pairs from $\{0, 1, 3, 7\}$, the distance between one pair, `max`(*pair1*) − `min`(*pair1*), is different from the distance between the other, `max`(*pair2*) − `min`(*pair2*). We believe that the reader will concur that the Essence specification closely matches the given English description, and is a substantially closer match than a standard CSP model of the problem.

The GRP specification first declares parameter n and then uses a `where` statement to constrain it to be positive. If the condition of a `where` statement is not met, then the input values are not valid; that is, they do not define an instance of the problem. The `where` statement could have been avoided by changing the `given` statement to `given` n: `int {1..}`. This fact is noted in a comment; comments always begin with a dollar sign and continue to the end of the line. The `letting` statement declares the identifier *bound*. It uses n, so n must be declared first. Identifiers must be declared before use, preventing cyclical definitions.

Constraints are built from parameters, constants, quantified variables and decision variables using operators commonly found in mathematics. Essence also includes variable binders $\forall x$, $\exists x$ and Σx, where x can range over any specified finite domain (e.g. a finite integer range but not all integers). The GRP constraint can be read "For any two unordered pairs of ticks, *pair1* and *pair2*, if the two pairs are different then the distance between *pair1* is not equal to the distance between *pair2*."

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*. The parameter *demand* is to be instantiated with a set of sets, where each inner set has 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 versions of a specification of the Social Golfers problem (SGP). As the problem description does not refer to the golfers individually, they are specified naturally with an unnamed type — that is, a type whose size is known but whose members have no names. The decision variable is represented straightforwardly as a set of partitions each representing a week of play. The specifications differ only in the expression of the socialisation constraint. The first constraint quantifies over the weeks, ensuring that the size of the intersection between every pair of parts of the corresponding partitions is at most one (otherwise the same two golfers are in a group together more than once). The alternative constraint quantifies over all pairs

of golfers, counting the number of times that each pair appears together and constraining this count to be strictly less than 2.

3 A Description of Essence

This section provides a fairly thorough description of Essence. The first subsection considers two preliminary issues: (1) the typography of the language — that is, how it appears; and (2) how Essence is processed, which has a profound effect on the design of the language. The second subsection discusses the distinction between types and domains. We then turn our attention to a sequence of subsections on particular features: types, domains, the universe of computation, specifications, expressions, and instantiation.

Throughout the paper τ (possibly with a subscript or prime) denotes an arbitrary type, ω (possibly with a subscript of prime) denotes an arbitrary ordered type (as explained later), and $exp{:}\tau$ or $e{:}\tau$ denotes an expression of type τ.

3.1 Preliminaries

Lexicography Abstractly, an Essence specification is a sequence of lexemes. Concretely, these lexemes can take different forms for different purposes. In this paper they take a form suitable for typesetting. In another variant of the language each lexeme is a sequence of characters commonly found on computer keyboards. In both these variants there is a one-to-one mapping from the concrete lexemes to the abstract lexemes; hence there is a one-to-one mapping between the two variants. One could imagine other variants designed for input from other devices or for displaying on other media.

As the typeset variant of Essence is used in this paper, we shall describe that. Each lexeme is either a keyword (such as `letting`, `int` or `size` in the GRP) a special symbol (such as "$\nmid$", "$\forall$", or "$\neq$" of the GRP), or a user-defined identifier (such as *pair1*, *bound*, or *Ticks* in the GRP). Every keyword or user-defined identifier is a sequence of letters and/or digits beginning with a letter. In addition, a keyword or identifier may end with a sequence of primes and/or a comma-separated sequence of subscripts, where each subscript is a sequence of letters and/or digits. In all cases the letters may be upper-case or lower-case, with different cases considered to be different characters. The font in which a keyword or identifier is written is immaterial, as is the spatial layout of the specification. The specifications of Fig. 1 are laid out in columns and use a variety of fonts; these are merely typesetting conventions used to enhance the appearance of the specifications.

Processing Stages and Instantiation Categories The architecture for processing a constraint language determines when parameters are instantiated, which, in turn, can place demands on the design of the language. This issue runs through most of the design, so it must be considered first.

There are two fundamentally different architectures for processing a constraint language — be it a problem specification language or a modelling language. One architecture is to first instantiate the parameters of the problem specification to obtain a specification of a single problem instance. The problem instance is then translated into the language of the solver and finally it is solved. We call this the instantiate-translate-solve (ITS) architecture. The alternative architecture translates the specification before instantiating it, and is therefore called the translate-instantiate-solve (TIS) architecture.

As our primary intended use of Essence is automated modelling, the language must support the use of TIS processing. Human constraint modellers almost always model entire problem classes, not single instances. The model for the problem class usually takes the form of a set of parameterised constraints; instantiating the parameters produces the constraints that model the corresponding problem instance. In other words, human modellers typically employ TIS processing; therefore, Essence must enable automated TIS processing. To be clear, by enabling TIS processing Essence also enables ITS processing; building an ITS system is easier than building a TIS system.

The TIS architecture places demands on the language design beyond those required by the ITS architecture. In particular, it must be possible to determine the grammaticality of a specification and translate it even though the values of the parameters are unknown.

Every expression in Essence belongs to one of three *instantiation categories*. *Constant expressions* contain no parameters, no decision variables and no unbound quantified variables. Thus a constant expression has the same value in every instance of the problem and that value can be determined during translation. *Parameter expressions* may contain parameters but contain no decision variables or unbound quantified variables. Hence the value of a parameter expression can be determined during instantiation. Though this value can vary from instance to instance, in a particular instance it has the same value in every solution (indeed, in every assignment to the decision variables). Finally, *variable expressions* can contain decision variables and unbound quantified variables. Hence the value of a variable expression is generally determined during the solution stage and can vary between the solutions to an instance. Observe that every constant expression is a parameter expression and every parameter expression is a variable expression.

To illustrate these definitions, notice in the GRP specification of Fig. 1 that 0 is a constant expression; n, $2^{**}n$ and *bound* are parameter expressions; and $Ticks$ and $\texttt{max}(Ticks)$ are variable expressions. Also notice that $\sum i \in \{1,2,3,4\} \,.\, 2^{**}i$ is a constant expression because i is a bound quantified variable, but $2^{**}i$ is only a variable expression because i is an unbound quantified variable

3.2 Types versus Domains

Essence is a strongly-typed language; every expression has a type (independently of where it occurs), and the types of all expressions can be inferred and

checked for correctness. Types are also important in determining the denotation of an overloaded operator. For example, the union operator can denote set union or multiset union depending on the types of its operands.

Essence is a finite-domain language; every decision variable is associated with a finite domain of values. These domains can be quite intricate sets of values. For example a domain could be any finite subset of the integers or it could be the set of two-element sets drawn from a given finite set of integers.

Types and domains play a similar role; they prescribe a range of values that a variable can take. It is tempting — and, indeed, we were tempted — to view types and domains as one and the same thing. However, this view leads to the difficult, if not unsolvable, problem that the intricate patterns used in constructing domains must be handled by the type system. For example, if every finite subset of the integers is a distinct type then the problem of assigning types to expressions is difficult, if not impossible.

As we have desired to keep the type system of Essence simple, we have reached the decision that types and domains are distinct — though closely related — concepts. Types denote non-empty sets that contain all elements that have a similar structure, whereas domains denote possibly empty sets drawn from a single type. In this manner, each domain is associated with an underlying type. For example integer is the type underlying the domain comprising integers between 1 and 10; set of integers is the type underlying the domain comprising all sets of two integers between 1 and 10. Type checking, type inference, and operator overloading are based only on types, not on domains.

Distinguishing types and domains helps resolve another problem. In many — perhaps almost all — combinatorial problems the combinatorial object sought varies in size between instances of the problem. Thus the domain of the decision variable(s) is usually parameterised, allowing it to vary from instance to instance. For example, in all four specifications of Fig. 1 the `find` statement specifies a domain containing parameters. In contrast, as types are used during the translation stage to determine the grammaticality of a specification, they are not parameterised. Distinguishing types and domains has enabled us to design Essence so that types can be determined before instantiation but domains can vary between instances.

Finally types and domains play different roles in our formal grammar of the language. The grammar is specified using a typed BNF notation whose types are precisely those of the language. Whereas types are meta-linguistic constructs that do not explicitly enter the language, domains do appear as strings in the language and are introduced by the non-terminal *Domain* of the formal grammar.

3.3 Types

Our design goal has been to provide Essence with a rich collection of types, yet a simple type system. The richness of the types comes from the large number of types and type constructors that are supported. The simplicity of the type system stems from several factors: every expression has a unique

type independently of where it occurs, these types can be determined in the translation stage (i.e. they are independent of parameter values), there is no subtyping and there is no type coercion.

The atomic types of Essence are `int` (integer), `bool` (Boolean), all user-defined enumerated types and all user-defined unnamed types. Enumerated types can be defined through `letting` or `given` statements and unnamed types can be defined through `letting` statements:

`letting` *players* `be new type enum` {*alan*, *berna*, *chris*, *ian*} (1)
`letting` *rings* `be new type of size` n (2)
`given` *players* `new type enum` (3)

Statement (1) defines a new type comprising four named atomic elements. These four names cannot be used to name anything else in the specification. Statement (2) defines a new type comprising n elements. These elements are not named. The size of the unnamed type can be specified by any parameter expression. Statement (3) is similar to (1) except that the enumeration of the elements of the type is provided as input rather than as part of the specification. Once again, the names provided cannot be used to name anything else.

The elements of the integer and Boolean types and of all enumerated types are totally ordered. The integers are ordered in the usual way, the Booleans are ordered by *false* $<$ *true*, and the elements of an enumerated type take on the order in which they are named in the `letting` statement or given as input. All other types — user-defined unnamed types and all compound types — are unordered.

Compound types are built with type constructors to form sets, multisets, functions, tuples, relations, partitions and matrices. Fig. 2 shows every type constructor of Essence along with its approximate denotation. The denotation is only approximate due to a subtlety that is addressed in Sec. 3.5.

Constructor	Denotation (approximate)
`set of` τ	every finite subset of τ
`mset of` τ	every finite multiset drawn from τ
`function` $\tau_1 \longrightarrow \tau_2$	every finite partial function with domain τ_1 and codomain τ_2
`tuple` $\langle \tau_1, \ldots, \tau_n \rangle$	$\tau_1 \times \cdots \times \tau_n$
`rel of` $(\tau_1 \times \cdots \times \tau_n)$	every finite subset of $\tau_1 \times \cdots \times \tau_n$
`partition from` τ	every partition of every finite subset of τ
`matrix indexed by` $[\ \omega_1, \ldots, \omega_n\ ]$ `of` τ	every n-dimensional matrix such that each dimension i is indexed by some finite range of values of type ω_i and each matrix entry is a member of τ.

Fig. 2. The type constructiors of Essence.

To illustrate the table, observe that `set of int` and `rel of` (`int` $\times$ `int`) are both types. The type constructors can be nested to arbitrary depth, thus allowing types such as

`set of set of set of int` (4)

`rel of (partition from` *players* $\times$ `mset of set of int`) (5)

The type names have been constructed so that they are unambiguous. Nonetheless, parentheses can be added to increase readability.

In normal English usage, a "partition of players" is a partition in which every player participates. In Essence a "`partition from` *players*" is a partition in which every participant is of type *players*. In other words, as stated in the table, a "`partition from` *players*" is a partition *of* some *subset* of *players*.[2] Essence highlights this distinction by using the word "from" rather than the word "of."

The phrase "of α" in a type name indicates that each value in the type is composed of values drawn from α. In contrast, the values in the types "`function` $\tau_1 \longrightarrow \tau_2$" and "`tuple` $\langle \tau_1, \ldots, \tau_n \rangle$" are precisely $\tau_1 \longrightarrow \tau_2$ and $\tau_1 \times \cdots \times \tau_n$.

Having considered how types can be constructed, let us consider their denotation. As one would expect, each type denotes a set of combinatorial objects. Notice that finiteness[3] plays a central role in the semantic explanation accompanying each type constructor above. A consequence is that, although types may contain an infinite number of elements, each element is of a finite size or cardinality. This is necessary to achieve the objective that Essence specifications can be mapped to CSPs.

Intuitively, a two-dimensional matrix is just another way of thinking about a one-dimensional matrix in which each entry is itself a one-dimensional matrix. In Essence we consider any type of the form

`matrix indexed by` $[\omega_1, \ldots, \omega_n]$ `of matrix indexed by` $[\omega_{n+1}, \ldots, \omega_m]$ `of` τ

to be isomorphic to `matrix indexed by` $[\omega_1, \ldots, \omega_m]$ `of` τ.

Finally notice that parameters, decision variables and quantified variables do not enter the type names.[4] This restriction is necessary to enable the type of every expression to be determined before the parameters are instantiated with values. This is why we say that Essence is statically typed.

3.4 Domains

A domain is a set of values all of the same type. In Essence every type is a domain. Essence also allows domains to be named by annotating the name of the type with restrictions that select particular values of the type. For example, `int` $\{1..10\}$ and `set` $\{$`size 2`$\}$ `of int` $\{1..10\}$ are both domains. As annotations are always written in dedicated annotation brackets, the type underlying a domain can always be obtained by removing all subexpressions contained within annotation brackets. Thus, the types underlying the above two domains are `int` and `set of int`, respectively. Each annotation that is added to a domain further restricts the elements that belong to the domain.

[2] It is worth noting that the empty partition is a partition of the the empty set of players.

[3] A function is finite if it is defined on only a finite set of values. Similarly, a relation is finite if it contains only a finite set of tuples.

[4] A parameter can appear in the declaration of a unnamed type, such as in statement (2), but it is not part of the type name.

First consider how atomic types can be annotated to form atomic domains. Atomic domains can be formed by taking subsets of the integer or Boolean type or of any enumerated type. These subsets are identified either by a list containing values and value ranges or by an arbitrary set expression. Examples include:

players {*alan..chris*}	(the players from *alan* to *chris* inclusive)	(6)
players { }	(the empty domain of players)	(7)
`int` {1, 3, 4..10}	(a mixture of values and ranges)	(8)
`int` {1..}	(the positive integers)	(9)
`int` {..-1,1..}	(the non-zero integers)	(10)
`int` {..}	(all integers; a degenerate case)	(11)
`int` {*l..u*}	(the integers from l to u inclusive)	(12)
`int` {$S \cup \{0,1\}$}	(the integers in S and 0 and 1)	(13)

In domains (12) and (13) l, u and S are intended to be parameters. The meaning of an expression of the form "*l..u*" is $\{x \mid l \leq x \leq u\}$. Hence, if $l > u$ then "*l..u*" denotes the empty set.

In these examples, and indeed all examples of domains, any of the constants could be replaced by any parameter expression of the same type. However, domains can never contain decision variables or unbound quantified variables. Observe that in CSPs, the domains associated with a decision variable can vary from instance to instance but they must be fixed within an instance – that is, a problem can never involve finding the elements of the domain. This has been reflected in the design of Essence because specifications are to be mapped to CSPs.

Now consider how type constructors can be annotated. The set, multiset, partition, function and relation type constructors can all be annotated by inserting, after the initial keyword, a size restriction of the form {`size` *exp*:`int`}, {`maxSize` *exp*:`int`} or {`minSize` *exp*:`int`}. Indeed, a `maxSize` and `minSize` restriction can be imposed together. In addition, the multiset type constructor may also have occurrence annotations of the form {`maxOccur` *exp*:`int`} or {`minOccur` *exp*:`int`}, which bound the number of occurrences of each value. Thus, all the following are domains:

`mset` {`maxOccur` 5} `of int`	(14)
`set` {`minSize` m, `maxSize` n} `of int`	(15)
`mset` {`maxOccur` 5, `size` n} `of int`	(16)
`mset` {`maxSize` $n+2$} `of int` {1..100}	(17)
`partition` {`size` n } `from mset` {`maxSize` 4} `of int` {1..100}	(18)

Domains (15) and (16) illustrate multiple annotations on a single type constructor. Domains (17) and (18) illustrate that the annotations can be attached not only to the outer constructor, but also to the nested constructors and atomic types. The Essence syntax has been designed so that there is no ambiguity about where an annotation is attached.

The function type constructor can have one annotation to indicate that the function is total or partial and another to indicate that the function is surjective, injective or bijective. For example, the following are domains:

`function` {`total`} *players* $\longrightarrow$ *players*	(19)
`function` {`injective`} *players* $\longrightarrow$ *players*	(20)

```
function {total} int ⟶ int                          (21)
function {total} int {1..10} ⟶ int                  (22)
function {total} int {1..10} ⟶ int {1..10}          (23)
```

Notice that domain (21) is empty as there are no finite functions that are total over an infinite domain. However, domain (22) is infinite and domain (23) is finite.

The tuple type constructor takes no annotations, though, of course, the types nested within it can be annotated, as in this example:

```
tuple ⟨ int {1..}, set {size 2} of int ⟩            (24)
```

The matrix type constructor takes no annotation itself, but the types of its indices can each be annotated. There are two kinds of domains that one can build using the matrix type constructor: *definite* and *indefinite*. In definite domains the matrix type constructor must specify completely its index values by providing a single finite range, as in these examples.

```
matrix indexed by [int {5..10}] of bool             (25)
matrix indexed by [players {berna..}] of bool       (26)
matrix indexed by [players {..chris}] of bool       (27)
matrix indexed by [players {..}] of bool            (28)
```

As previously seen with ranges, (26) is a matrix indexed by all *players* greater than or equal to *berna*, (27) is indexed by all *players* less than or equal to *chris* and (28) is indexed by all players. A domain is definite if every index of all its matrix type constructors are definite.

Here are three examples that are *not* definite domains:

```
matrix indexed by [int {1..}] of bool               (29)
matrix indexed by [players] of bool                 (30)
matrix indexed by [int {1..10}, players] of bool    (31)
```

The first example, (29), is not a definite domain because it has an infinite set of index values. This is disallowed because matrices must have a finite size. Example (30) is not a definite domain because *players* is not annotated with a range; hence it represents all possible subsets of *players*. Finally, consider example (31). Although the first index of the matrix is definite, the second is not; thus (31) is not a definite domain.

An Essence specification associates a finite domain with each decision variable, and this domain must be definite. Thus, solving a problem instance can require finding values for the entries of a matrix, but it cannot involve finding the index values of a matrix.

An Essence specification associates a domain with each parameter, and this domain can be definite or indefinite. If it is definite then it specifies precisely what the index values of the input matrix must be. However, it may also be indefinite, providing only partial information or no information on what the index values of the input matrix must be. This is achieved either by omitting the annotation on the type of the index, as in examples (30) and (31), or by using an underscore as one or both of the index bounds, as in these examples:

```
matrix indexed by [int {1.._}] of bool              (32)
matrix indexed by [int {_..100}] of bool            (33)
```

```
matrix indexed by [int {_..}] of bool                    (34)
matrix indexed by [players {..}] of bool                 (35)
```

Example (32) denotes the set containing every finite matrix of Booleans whose smallest index value is 1 and whose largest index value is at least 1. Example (33) denotes the set containing every finite matrix of Booleans whose largest index value is 100 and whose smallest index value is at most 100. Example (34) denotes the set containing every finite matrix of Booleans indexed by integers; it is thus synonymous with `matrix indexed by [int] of bool`. Finally, (35) shows that an omitted bound can be combined with an underscore. This indefinite domain denotes the set containing every finite matrix of Booleans whose smallest index is the least value of *players* (that is, *alan*) and whose largest index is greater than or equal to *alan*. Finally, it is worth observing that (29) is neither a definite nor an indefinite domain since it denotes matrices with an infinite set of index values.

As we have seen, the partition type constructor can be annotated with the usual size annotations; these constrain the number of elements within all the parts (sets) in the partition. In addition, there are three annotations to constrain the size of the parts within the partition — `partSize`, `maxPartSize` and `minPartSize` — and three to constrain the number of parts within the partition — `numParts`, `maxNumParts` and `minNumParts`. The partition type constructor can also have an annotation to indicate that the partition is regular; that is, all of its parts are of the same size. Some of these combinations are illustrated by these examples:

```
partition {size 32} from int{1..32}                      (36)
partition {size p, numParts n, regular} from int {1..2 * p}   (37)
partition {numParts n, minSize m} from players           (38)
partition {partSize m} from int{1..31}                   (39)
```

Notice that domain (37) is empty if p is not a multiple of n; otherwise the partition contains n parts each of size p/n. Also notice that the denotation of domain (39) is a set of regular partitions since every set in the denotation must have size m.

Recall that the domain `partition from` *players* contains every partition of every subset of *players*. The annotation `complete` can be used to restrict the domain to contain only partitions in which all *players* participate. Thus the `complete` annotation for partitions is analogous to the `total` annotation for functions. The analogy is realised by considering a partition to be a function from potential participants to groups within the partition. Consider some examples:

```
partition {complete} from int{1..32}                     (40)
partition {complete} from int{1..}                       (41)
```

Domains (36) and (40) are equivalent since a complete partition from 1..32 must involve all 32 of these integers. Domain (41) is empty since any finite partition of the positive integers cannot involve all the positive integers. Thus this domain and domain (21) are empty for similar reasons.

3.5 Types and the Universe of Computation

A central concern for the semantics of a language is to identify the universe, that is all the objects that can be talked about by the language. In Essence we would like the universe to be precisely those objects that can partake in a solution (i.e., a satisfying assignment). Since the value assigned to a decision variable must be drawn from its associated domain, the universe must be precisely the union of the denotations of all finite, definite domains. And since all members of a domain are members of its underlying type, the denotation of a type τ must be the union of the denotations of all finite, definite domains whose underlying type is τ.

All of the above considerations are met if one assigns denotations to types and domains in a straightforward manner — but there is one exception. Using the straightforward meaning of types suggested in Fig. 2 the type

`set of matrix indexed by` [`int`] of *players* (42)

contains every finite subset of the set containing every one-dimensional matrix of *players* that is indexed by any finite range of integers. Thus, a literal reading of the table says that a member of type (42) could be a set containing matrices that have different index values. However, every finite, definite domain with this underlying type denotes a set of matrices that have the same index values.

Therefore (42) cannot be given its straightforward interpretation; it must denote the union over every finite range r of integers of $M(r)$, where $M(r)$ is the set containing every finite subset of the set containing every one-dimensional matrix of *players* indexed by r. Another way to view this is by considering a pseudo matrix type constructor that specifies the exact range of the matrix. Using this pseudo constructor, type (42) is equivalent to

$$\bigcup_{r:\text{finite range of } \texttt{int}} \texttt{set of matrix indexed by } [r] \text{ of } \textit{players} \tag{43}$$

This semantic treatment of the matrix type constructor extends to deeper nesting. For example type (44) can be seen as being equivalent to pseudo type (45).

`set of set of matrix indexed by` [`int`] of *players* (44)

$\bigcup_{r:\text{finite range of } \texttt{int}}$ `set of set of matrix indexed by` [r] of *players* (45)

For a type with multiple matrix type constructors each constructor can independently fix its index values, Thus type (46) can be seen as being equivalent to pseudo type (47).

`set of tuple` ⟨ `matrix indexed by` [`int`] of *players*, `matrix indexed by` [`int`] of *players* ⟩ (46)

$\bigcup_{r,r':\text{finite range of } \texttt{int}}$ `set of tuple` ⟨ `matrix indexed by` [r] of *players*, `matrix indexed by` [r'] of *players* ⟩ (47)

A consequence of the treatment of the matrix type constructor is that the type

`matrix indexed by` [`int`] of `matrix indexed by` [`int`] of *players* (48)

allows the outer matrix (say, the rows) to be indexed by one range and the inner matrix (say, the columns) to be indexed by another range, but each row must be of the same size and have the same index values. Thus, the elements of type (48) are all rectangular two-dimensional matrices; so type (48) can be seen as equivalent to (49).

`matrix indexed by` [`int`, `int`] of *players* (49)

We have illustrated this issue with examples in which a matrix type constructor is nested within a set or matrix type constructor. The same issue arises (and the same approach used) when a matrix type constructor is nested within any type constructor except the tuple type constructor.

3.6 Specifications and Statement Merging

The syntax of Essence has three principal components: domains, which were explained in the previous subsection; specifications, which are explained in this subsection; and expressions, which are explained in the next subsection. The grammar of the language contains three principal non-terminals, one for each of these components.

As explained in Section 2, Essence has eight kinds of statements. A specification is a list of statements composed according to the regular expression:

`language` (`given` | `letting` | `where` | `find`)* [`minimising` | `maximising`] (`such that`)*

For ease of presentation, we shall explain the language in terms of individual statements each doing a single thing such as declaring a single identifier or imposing a single constraint. In addition to supporting these single statements, Essence allows multiple statements of the same kind to be merged together. For example, the Mystery Problem of Fig. 1 contains a `given` statement that could be seen as the merge of five separate `given` statements and a `such that` statement that could be seen as the merge of two separate `such that` statements. Additionally, `where`, `letting` and `find` statements can each be merged in a similar manner.

The `language` statement is self-explanatory. We now describe the other statements, including two forms of the `given` statement and four forms of the `letting` statement. Here *NewID* is a new identifier — that is, one which has not been declared before and which is not a built-in keyword or identifier; *Domain* is an arbitrary domain.

`given` *NewID* : *Domain*
: This statement declares *NewID* to be a parameter. Its value is given by the input and must be a member of *Domain*. Subsequently, *NewID* can be used as a parameter expression whose type is the type that underlies *Domain*.

`given` *NewID* `new type enum`
: As explained in Section 3.3, this statement declares a new enumerated type whose elements are given by the input. Thus the elements of this type can vary from problem instance to problem instance.

`letting` *NewID* `be` *exp*

In this statement *exp* can be any parameter expression. This statement declares *NewID* to be an identifier whose value, type and category are those of *exp*. Subsequently, *NewID* can be used as an expression of that type and category.

`letting` *NewID* `be domain` *Domain*

This statement declares *NewID* to be a new identifier which can be used to stand for any domain *Domain*.

`letting` *NewID* `be new type of size` *exp*:`int`
`letting` *NewID* `be new type enum` $\{NewID_1, \ldots, NewID_n\}$

As explained in Section 3.3, these two statements declare a new unnamed type and a new enumerated type, respectively.

`where` *exp*:`bool`

This statement is used to constrain the input values that can be bound to parameters; hence the constraint, *exp*:`bool`, must be a parameter expression. Input values that do not satisfy the constraint do not define an instance of the problem.

`find` *NewID* : *Domain*

This declares *NewID* to be a finite-domain decision variable whose type is the underlying type of *Domain*. The domain of the decision variable is the set of values denoted by *Domain*, which must be finite and definite.

`minimising` *exp*:ω
`maximising` *exp*:ω

These statements define the objective function in an optimisation problem.

`such that` *exp*:`bool`

This statement is used to impose a constraint that a solution must satisfy. A constraint is simply an expression of type `bool`, which — unlike a `where` statement — can, and usually does, contain decision variables.

3.7 Expressions

The expressions of Essence are formed in a manner much as one would expect, bearing in mind that every expression has a type. We first consider atomic expressions and then compound expressions.

Atomic Expressions The atomic expressions of Essence are the atomic constants, parameters, decision variables and quantified variables. Other than unnamed types — and compound types constructed from them — every value has a name. Integers are named in the usual manner and the Booleans are named *true* and *false*. The values of compound types are named by using value constructor operators, so these are explained in the subsequent discussion of compound expressions.

There is one other kind of expression that is considered to be atomic. If d is a finite, definite domain then writing d within straight double quotation

marks — that is, "d" — is an expression that denotes the set containing all elements of the domain d. Since domains are not expressions, "d" can not be construed as a quotation operator applied to an expression, which is why we do not consider it to be a compound expression.

Compound Expressions Compound expressions are formed by composing together an operator with other expressions, which are its operands. The denotation of a compound expression is obtained by applying the operation denoted by the operator to the denotation of its operands. Thus "applying an operator" shall refer to both the process of forming a compound expression and to obtaining the denotation of that expression. And when we say that an operator is applied to, say, an integer, we refer both to forming a compound expression from the operator and an integer expression, and to obtaining a value by applying the operation denoted by the operator to the integer denoted by the integer subexpression.

In general, the operands of an operator can be arbitrary expressions, including decision and quantified variables, of the appropriate type. In the presentation that follows, this is assumed unless stated otherwise.

Some operators are overloaded. For example, the intersection operator can be applied to two multisets to produce a multiset and it can be applied to two sets to produce a set. With only two exceptions (both of which are explained below) the type of a compound expression is a function of its operator and the *types* of its operands.

There are two operators, equality and disequality, that can be applied to all types. For any type τ, both $exp_1{:}\tau = exp_2{:}\tau$ and $exp_1{:}\tau \neq exp_2{:}\tau$ are expressions of type `bool`.

We now consider the remaining operators in four categories: value construction operators, operators on atomic types, operators on set-like structures (sets, multisets, functions, relations and partitions) and operators on tuples and matrices.

Value Construction Operators Essence provides operators for constructing values for compound types from values for the constituent parts, for example to create a set from an explicit list of elements. These operators are given in Fig. 3, where $m, n \geq 1$ and if τ is a type then 'τ' is its name — that is, a string denoting the type τ.

In general, the type of the constructed object is a function of the type of the constituent parts; however for empty collections (empty sets, multisets, relations, functions and partitions) the type of the constituent parts is not available; hence in these cases — rows (2), (4), (6), (8) and (10) in Fig. 3 — the type must be provided.

Sets are constructed by providing the elements of the set (1). These elements are not required to be different, so that providing n elements results in a set of size less than n if there are duplicate values. Multisets are constructed by providing the elements of the multiset (3). Relations are constructed by providing the tuples that comprise the relation (5); as with sets, duplicates are

	Expression	Type
1)	$\{e_1{:}\tau, \ldots, e_n{:}\tau\}$	`set of` τ
2)	$\{\}$: '`set of` τ'	`set of` τ
3)	`mset`$(e_1{:}\tau, \ldots, e_n{:}\tau)$	`mset of` τ
4)	`mset`$(\,)$: '`mset of` τ'	`mset of` τ
5)	`rel`$(e_1{:}$`tuple` $\langle\tau_1, \ldots, \tau_m\rangle, \ldots,$ $e_n{:}$`tuple` $\langle\tau_1, \ldots, \tau_m\rangle)$	`rel of` $(\tau_1 \times \cdots \times \tau_m)$
6)	`rel`$(\,)$: '`rel of` $(\tau_1 \times \cdots \times \tau_m)$'	`rel of` $(\tau_1 \times \cdots \times \tau_m)$
7)	`function`$(e_1{:}\tau \mapsto e_1'{:}\tau', \ldots, e_n{:}\tau \mapsto e_n'{:}\tau')$	`function` $\tau \longrightarrow \tau'$
8)	`function`$(\,)$: '`function` $(\tau \longrightarrow \tau')$'	`function` $\tau \longrightarrow \tau'$
9)	`partition`$(e_1{:}$`set of` $\tau, \ldots, e_n{:}$`set of` $\tau)$	`partition from` τ
10)	`partition`$(\,)$: '`partition from` τ'	`partition from` τ
11)	$\langle e_1{:}\tau_1, \ldots, e_n{:}\tau_n\rangle$	`tuple` $\langle\tau_1, \ldots, \tau_n\rangle$
12)	$[e_1{:}\tau, \ldots, e_n{:}\tau]$	`matrix indexed by [int] of` τ
13)	$[e_1{:}\tau, \ldots, e_n{:}\tau \;;\; e{:}\omega]$	`matrix indexed by` $[\omega]$ `of` τ

Fig. 3. Value construction operators.

ignored. Functions are constructed by providing a mapping from each (function) domain element for which the function is defined to the corresponding range element (7). Duplicate mappings (where two mappings map the same domain element to the same range element) are permitted and ignored, but a domain element may not be mapped to more than one range element; moreover, we restrict the expressions appearing in a function constructor to be of category parameter so that this condition can be checked at instantiation time. Partitions are constructed by providing a set for each part of the partition (9). These sets must be disjoint, and again are restricted to be of category parameter. Tuples are constructed by providing the elements of the tuple (11).

For matrix construction ((12) and (13)) we need to know the index range of the matrix in addition to its entries. By default (12) the index of the matrix is assumed to be of type `int` with index values starting at 1 and finishing at n, where n is the number of provided elements. To construct matrices with indices of other types, or with index values starting at an integer other than 1, a second form (13) can be used, where the starting index value is provided by an additional expression, which must be of category parameter. The type of the index is taken from the type of this value, and the end value of the index range inferred from the start value and the number of provided elements.

Operators on Atomic Types Members of all ordered types can be compared with the usual inequality operators, $\leq$, $<$, $\geq$ and $>$. To Booleans one can apply the usual logical operators: $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and $\neg$. To integers one can apply the common arithmetic operators: $+$, $-$, $*$, $/$, %, **, and $|\cdot|$. The last three of these denote mod, exponentiation and absolute value, respectively.

The minus operator is both unary and binary. The operator `toInt` converts a Boolean to an integer by mapping *false* to 0 and *true* to 1.

Operators on Sets, Multisets, Functions, Relations and Partitions Sets, multisets, partitions, relations and functions are all set-like structures; thus there is considerable commonality among the operators that can be applied to them. Fig. 4 displays all operators that can be applied to these set-like structures. To illustrate how the table is to be read, consider row (1). This shows that the union operator can be applied to any two expressions of type τ provided that τ is of one of three forms: `set of` τ' for some τ'; `mset of` τ' for some τ'; or `rel of` $(\tau_1 \times \ldots \times \tau_n)$, for some $\tau_1, \ldots, \tau_n$. The table shows that in all three cases the expression $exp_1{:}\tau \cup exp_2{:}\tau$ is of type τ. The two blank boxes in that row indicate that the union operator cannot be applied to partitions or to functions.

The union operator illustrates the simple approach that has guided the definition of all of these operators. The union operator can only be applied to two operands of the same type. To union expressions of different types, one can apply conversion operators to make them the same type. For example, to union an expression S of type `set of` τ with an expression M of type `mset of` τ one could write `toMset`$(S) \cup M$, which converts S to a multiset using the `toMset` operator of row (6) of the table. Also notice that two partitions cannot be unioned because the result would not generally be a partition; similarly, two functions cannot be unioned. Such unions could be performed after first converting the two partitions or two functions to two sets.

Rows (2) through (6) of the table are for familiar set operators and require no explanation other than to point out that the the `min` and `max` operators of row (7) can be applied to an expression of type `set of` τ or `mset of` τ only if τ is an ordered type.

Rows (7), (8) and (9) are for operators that convert an operand to a set, multiset or relation. Only functions can be converted to relations, but anything other than a set or a partition can be converted to a set and anything other than a multiset or a partition can be converted to a multiset. For example, the `toSet` operator maps a multiset to a set that contains precisely those elements that occur in the multiset and `toMset` maps a set to a multiset that has exactly one occurrence of each element in the set. The inverses of these conversion operators can be obtained by imposing equational constraints. For example to convert relation R to a function F one could impose `toRel`$(F) = R$, and to convert a set S of tuples to relation R one could impose `toSet`$(R) = S$.

Rows (10) through (15) are are for operators on functions. The `defined` operator of row (10) obtains the set of values on which a function is defined and the `range` operator of row (11) obtains the set of values to which the defined values are mapped. Row (12) is for the usual function application: If F is a function then $F(exp)$ and `image`(F, exp) are synonymous and denote the result of applying F to exp. The `image` operator can also be used to apply a function to a set S producing the set $\{F(s)|s \in S\}$, as given in row (13).

	τ is set of τ'	τ is mset of τ'	τ is partition from τ'	τ is rel of $(\tau_1 \times \cdots \times \tau_n)$	τ is function $\tau_1 \longrightarrow \tau_2$
1) $e_1{:}\tau \cup e_2{:}\tau$	τ	τ		τ	
2) $e_1{:}\tau \cap e_2{:}\tau$ $e_1{:}\tau - e_2{:}\tau$	τ	τ		τ	τ
3) $e_1{:}\tau \subset e_2{:}\tau$ $e_1{:}\tau \subseteq e_2{:}\tau$ $e_1{:}\tau \supset e_2{:}\tau$ $e_1{:}\tau \supseteq e_2{:}\tau$	bool	bool		bool	bool
4) $\lvert e{:}\tau \rvert$	int	int	int	int	int
5) $e'{:}\tau' \in e{:}\tau$	bool	bool			
6) max$(e{:}\tau)$ min$(e{:}\tau)$	τ' †	τ' †			
7) toSet$(e{:}\tau)$		set of τ'		set of tuple $\langle \tau_1, \ldots, \tau_n \rangle$	set of tuple $\langle \tau_1, \tau_2 \rangle$
8) toMset$(e{:}\tau)$	mset of τ'			mset of tuple $\langle \tau_1, \ldots, \tau_n \rangle$	mset of tuple $\langle \tau_1, \tau_2 \rangle$
9) toRel$(e{:}\tau)$					rel of $(\tau_1 \times \tau_2)$
10) defined$(e{:}\tau)$					set of τ_1
11) range$(e{:}\tau)$					set of τ_2
12) $e{:}\tau(e_1{:}\tau_1)$ image$(e{:}\tau, e_1{:}\tau_1)$					τ_2
13) image$(e{:}\tau, e_1{:}$set of $\tau_1)$					set of τ_2
14) preimage$(e{:}\tau, e_2{:}\tau_2)$					set of τ_1
15) inverse$(e{:}\tau,$ $e'{:}$function $\tau_2 \longrightarrow \tau_1)$					bool
16) $e{:}\tau(e_1{:}\tau_1, \ldots, e_n{:}\tau_n)$				bool	
17) $e{:}\tau(\ldots, \_, \ldots)$				rel of $(\tau_{i_1}, \ldots, \tau_{i_k})$	
18) together$(e_1{:}\tau', e_2{:}\tau', e{:}\tau)$ apart$(e_1{:}\tau', e_2{:}\tau', e{:}\tau)$			bool		
19) party$(e'{:}\tau', e{:}\tau)$ participants$(e{:}\tau)$			set of τ'		
20) parts$(e{:}\tau)$			set of set of τ'		
21) freq$(e{:}\tau, e'{:}\tau')$		int			
22) hist$(e{:}\tau,$ e':matrix indexed by $[\omega]$ of $\tau')$		matrix indexed by $[\omega]$ of int			

† provided τ is an ordered type.

Fig. 4. All operators on sets, multisets, partitions, relations and functions.

The `preimage` operator of row (14) is a kind of inverse of the `image` operator: `preimage`(F, v) denotes the set $\{y|F(y) = v\}$. The `inverse` operator of row (15) is true if its two operands are functions that are inverses of each other.

Rows (16) and (17) are for relation application and relation projection, respectively. If R is an expression of type `rel of` $(\tau_1 \times \cdots \times \tau_n)$ then $R(exp_1{:}\tau_1, \ldots, exp_n{:}\tau_n)$, is true if and only if $\langle exp_1{:}\tau_1, \ldots, exp_n{:}\tau_n\rangle$ is in relation R. One can also take a projection of a relation. For example, if R is an expression denoting a relation of arity 3, then $R(a, b, \_)$ denotes a unary relation containing every tuple $\langle c\rangle$ such that $R(a, b, c)$. Furthermore, $R(\_, b, \_)$ denotes the binary relation containing every tuple $\langle a, c\rangle$ such that $R(a, b, c)$. In general, these projection expressions must contain at least one argument that is an underscore and at least one argument that is not. The relation denoted by the expression has arity equal to the number of underscores.

Rows (18) through (20) are for operations on partitions. Let us first present some terminology. A partition is a collection of disjoint sets, each of which is called a part of the partition. The union of the parts is the set of participants of the partition. The party of participant p is the part that contains p. Let P be an expression of type `partition from` τ' and e_1 and e_2 be expressions of type τ'. Then `together`(e_1, e_2, P) denotes *true* if e_1 and e_2 are within the same part within P and `apart`(e_1, e_2, P) denotes *true* if e_1 and e_2 are within different parts of P. If either or both of e_1 and e_2 are not participants of P, then `together` and `apart` are both *false*. The expression `party`$(e'{:}\tau', P)$ denotes the part of P containing p. The expression `participants`(P) denotes the set containing all the participants of P.

Row (21) is the `freq` operator, which gives the number of occurrences of a value in a multiset. Row (22) is for a related operator: `hist`(*target*,*values*) generates a *histogram*, where *target* is a multiset of elements of type τ', *values* is a matrix with entries of type τ', and *histogram* is a matrix with integer entries. Furthermore, the index values of *values* and *histogram* must be identical, thus putting these matrices in a one-to-one correspondence. Then for every index i, the number of occurrences of *values*$[i]$ in *target* is *histogram*$[i]$.

Operators on Tuples and Matrices The entries of a tuple can be accessed by an index. If T is an expression of type `tuple` $\langle\tau_1, \ldots, \tau_n\rangle$ and i is an integer expression then $T\langle i\rangle$ denotes the i^{th} entry of T. In this case the type of $T\langle i\rangle$, which is τ_i, depends on the value of i. Thus to enable the type to be determined during the translation stage of the TIS scheme, Essence requires i to be of category constant and its value to be between 1 and n inclusively. Following common practice for matrices, if T is a nested tuple (a tuple containing a tuple, or a tuple containing a tuple containing a tuple, etc.) then one can write $T\langle i_1, \ldots, i_n\rangle$ as shorthand for $T\langle i_1\rangle \cdots \langle i_n\rangle$.

As one would expect, matrices can also be accessed by indices. If M is an expression of type `matrix indexed by` $[\omega_1, \ldots, \omega_n]$ `of` τ then $M[exp_1{:}\omega_1, ..., exp_n{:}\omega_n]$ is an expression of type τ. In contrast to tuple indexing, each index $exp_i{:}\omega_i$ can be an expression of any category.

In addition to indexing by values, ESSENCE also allows indexing by ranges to select a submatrix. Thus, if M_1 and M_2 are respectively one- and two-dimensional matrices, then $M_1[1..5]$ is a one-dimensional matrix containing 5 entries from M_1, $M_2[1..5, 2..4]$ is a two-dimensional matrix containing 5 rows and three columns selected from M_2, and $M_2[1, 2..4]$ is a one-dimensional matrix containing entries 2 to 4 from the first row of M_2. In general, the dimensionality of the expression is equal to the number of indices that use range expressions. The lower and upper bounds of the ranges can be any parameter expressions; variables expressions are disallowed so that the index values of the resulting matrix can be determined during the instantiation stage. Finally, the lower and/or upper bounds of an index range can be omitted, in which case they default to the lowest and highest indices of the matrix, respectively. Thus if M_1 is indexed from l to u then $M_1[a..]$ is shorthand for $M_1[a..u]$; $M_1[..b]$ is shorthand for $M_1[l..b]$; and $M_1[..]$ is shorthand for $M_1[l..u]$.

A matrix is said to have normal indices if each dimension is indexed by a range of integers starting at 1. The `normIndices` operator maps a matrix to a matrix that has normal indices but is otherwise identical. If `normIndices` is applied to nested matrices, then its action applies as deeply as possible. For example, if M is of type

`matrix indexed by [int,int] of matrix indexed by [int,int] of bool`,

then `normIndices`(M) behaves as if M is of type

`matrix indexed by [int,int,int,int] of bool`.

Note that if `normIndices` is applied to an expression of type `matrix indexed by [int] of tuple ⟨ matrix indexed by [int], bool ⟩` then its action applies only to the outermost matrix.

The `indices` operator is used for obtaining the index ranges of a matrix. If M is an expression of type `matrix indexed by` $[\omega_1, \ldots, \omega_n]$ `of` τ, where τ is not a matrix, then `indices`(M) is an expression of type `tuple` $\langle$`tuple`$\langle\omega_1, \omega_1\rangle, \ldots,$ `tuple`$\langle\omega_n, \omega_n\rangle\rangle$. For each dimension i of M, `indices`$(M)\langle i, 1\rangle$ is its smallest index value and `indices`$(M)\langle i, 2\rangle$ is its largest index value.

The `flatten` operator unrolls a matrix of any number of dimensions to produce a one-dimensional matrix with normal index values. The unrolling of a matrix selects the entries of the matrix according to the lexicographic ordering of their indices. For example if M is a matrix indexed by [`int` $\{1..2\}$, `int` $\{1..2\}$, `int` $\{1..2\}$] then `flatten`(M) denotes the matrix

$$[M[1,1,1],\ M[1,1,2],\ M[1,2,1],\ M[1,2,2],\ M[2,1,1],\ M[2,1,2],\ M[2,2,1],\ M[2,2,2]].$$

As with `normIndices`, if `flatten` is applied to nested matrices, its action applies as deeply as possible.

ESSENCE provides a more general version of `flatten` that enables one to specify the depth to which flattening is performed. If n is an integer expression of category constant and M is an expression of type `matrix indexed by`

$[\omega_1, \ldots, \omega_{n+1}]$ `of` τ then `flatten`(n, M) is of type `matrix indexed by` [`int`] of τ. The value of `flatten`(n, M) is obtained by flattening the first $n + 1$ dimensions of M into a single dimension with normal indices. Observe that this is the second case where the type of an expression depends on the *value* of an operand (see also tuple indexing above), which is why n must be a constant expression. Furthermore, its value must be positive.

ESSENCE has four lexicographic inequality operators — $\leq_{lex}$, $<_{lex}$, $\geq_{lex}$ and $>_{lex}$ — for comparing two expressions of type `matrix indexed by` $[\omega_1]$ `of` ω provided the two matrices have the same indices.

A matrix can be converted to a set or multiset containing the same values as the entries of the matrix. If M is an expression of type `matrix indexed by` $[\omega]$ of τ, then `toSet`(M) is of type `set of` τ and `toMset`(M) is of type `mset of` τ.

Three further operators can be applied to any expression M of type `matrix indexed by` $[\omega]$ `of` τ. An expression of the form `allDiff`(M) is true if all entries of M are different values. The operators `freq`(*target*,*value*) and `hist`(*target*,*values*) are exactly like those previously described (see Fig. 4) except that *target* is a matrix with any index values rather than a multiset. In fact, `freq`(*target*,*value*) is equivalent to `freq`(`toMset`(*target*),*value*) and `hist`(*target*,*values*) is equivalent to `hist`(`toMset`(*target*),*values*)

It should be observed that the operands of `allDiff`, `toSet` and `toMset` and the first operand of both `hist` and `freq` are specified as one-dimensional matrices. However each of these operands can also be, for example, a three-dimensional matrix M_3 since M_3 can be viewed as a one-dimensional matrix of two-dimensional matrices. Thus, for example, `allDiff`(M_3) is true if $M_3[1]$,...,$M_3[n]$ are all different two-dimensional matrices. To apply any of these operators to all entries $M_3[i, j, k]$ of M_3 one could write, for example, `allDiff(flatten`(M)`)`.

If ω is an ordered type then the `min` or `max` operator can be applied to any expression of type `matrix indexed by` $[\omega_1, \ldots, \omega_n]$ `of` ω. Like `flatten`, this operator applies as deeply as possible.

Quantified Expressions ESSENCE provides an exceptionally rich set of constructs for expressing quantification. Examples can be seen in all of the constraints of Fig. 1 as well as in the "`minimising`" statement of the SONET specification.

For ease of presentation, we present each quantifier as binding a single variable or structured variable. (Structured variables are explained later.) Akin to statement merging, ESSENCE allows two forms of quantification expression merging. An example of the first form of merging appears in the GRP specification of Fig. 1: the single quantification expression

$\forall$*pair1*, *pair2* : `set (size 2) of int` $\subseteq$ *Ticks*.

is the merger of two nested quantification expressions

$\forall$*pair1* : `set (size 2) of int` $\subseteq$ *Ticks*. $\forall$*pair2* : `set (size 2) of int` $\subseteq$ *Ticks*.

An example of the second form of merging appears in the SONET problem specification of Fig. 1: the single quantification expression

$\forall group1 \in \texttt{parts}(week1), group2 \in \texttt{parts}(week2).$

is the merger of two nested quantification expressions

$\forall group1 \in \texttt{parts}(week1). \forall group2 \in \texttt{parts}(week2).$

These mergers can also be performed with other forms of quantified expressions.

Each quantified expression comprises a sequence of four components:

- a quantifier, either $\sum$, $\forall$ or $\exists$;
- a structured variable, which is either a single variable or (as explained later) a compound structure, which identifies the variables that are being "declared";
- a binder that dictates a finite set of values over which the variables range. These values are all of the same type, and this determines the type of the associated quantified variables. The finiteness of the set of values over which a variable ranges is necessary to enable the mapping of specifications to CSPs.
- an expression, which must be of type `bool` if the quantifier is $\forall$ or $\exists$, and of type `int` if the quantifier is $\sum$. This expression and its sub-expressions are said to be within the scope of quantifier. This expression is separated from the preceding binder by a full stop.

We now survey the quantified expressions of Essence according to the four forms that binders can take. We consider each binder along with an ordinary variable that precedes it. Structured variables are considered after this survey.

The first binder form is to give a finite, definite domain, as in the universal quantifier of the alternative constraint of the SGP specification in Fig. 1.

The second form of binder draws elements from a set or multiset. This method is used by all the quantifiers in the Mystery Problem and the SONET Problem of Fig. 1. The set or multiset can be denoted by an arbitrary expression, including expressions that contain decision variables, as seen in the examples mentioned.

One should be careful with drawing elements from a mulitset. Consider:

$$\forall i \in \texttt{mset}(1,1,2).\ M[i] = i \tag{50}$$
$$\forall i \in \texttt{toSet}(\texttt{mset}((1,1,2)).\ M[i] = i \tag{51}$$
$$\sum i \in \texttt{mset}(1,1,2).\ i \tag{52}$$
$$\sum i \in \texttt{toSet}(\texttt{mset}(1,1,2)).\ i \tag{53}$$

Although (50) and (51) are equivalent, (52) and (53) are not; (52) denotes 4 but (53) denotes 3.

The third binder form draws all subsets from a set, all sub-multisets from a multiset, all sub-relations from a relation or all sub-functions from a function. Each of these can be either strict or non-strict. These correspond to the "$\subset$" and "$\subseteq$" operators of row (3) of Fig. 4. An example of this method is

$$x \subset (y \cap z) \tag{54}$$

The quantified variable x has the same type as $y \cap z$, be it set, multiset, relation or function. The binder can be an expression of any category.

The final binder form combines the first form with either the second or third form. An example is seen in the constraint of the GRP specification. Here *pair1* and *pair2* must each be a subset of $Ticks$ and must also be a set of size 2. In this binder form the explicit domain need not be finite or definite. In the GRP example, finiteness is obtained since the binding values must each be a subset of $Ticks$, which is itself finite; the domain, `set {size2} of int`, serves only to filter the set of potential binding values.

Structured variables enable binding to compound structures containing variables, as in these examples:

$$[x, y] : \texttt{matrix indexed by [int\{1..2\}] of int\{1..2\}} \tag{55}$$
$$[x, y] : \texttt{matrix indexed by [int\{1..2\}] of set of int\{1..3\}} \tag{56}$$
$$\langle x, y\rangle : \texttt{tuple } \langle \texttt{int\{1..2\}, bool}\rangle \tag{57}$$
$$[x, \langle y, z\rangle] : \texttt{matrix indexed by [int\{1..2\}] of tuple } \langle \texttt{int\{1..2\}, bool}\rangle \tag{58}$$

Example (55) binds $[x, y]$ to each element of the given domain by binding x and y to the appropriate values. In particular, there are four such bindings: x to 1 and y to 1; x to 1 and y to 2; x to 2 and y to 1; and x to 2 and y to 2. Thus x and y are both of type `int`. If x and y could not be bound to values that make $[x, y]$ take on the values in the given domain, then the expression is ill-typed. In example (56), x and y are of type `set of int`. The example of (57) shows that structured variables can be constructed with tuple constructors. No other value constructors can be used in structured variables, though matrix constructors and tuple constructors can be used together in a structured variable, as shown in example (58). In this example x is of type `tuple` $\langle$`int, bool`$\rangle$, y is of type `int` and z is of type `bool`.

Structured variables can be used with any binding form. In this example it is used with the second binder form:

$$\forall [x, y] \in S \tag{59}$$

This is well typed only if the type of S is of the form `set of matrix indexed by` $[\omega]$ `of` τ. If it is well-typed, then x and y are each of type τ.

Variable names cannot be repeated within a structured variable. Each quantified variable, within a structured variable or otherwise, must be a new identifier, although variable names introduced by quantifiers can be reused outside of their scope.

3.8 Instantiation

An instantiation of an Essence specification is provided by an instance file that gives a value for each parameter in the specification. There is a one-to-one correspondence[5] between the set of `given` statements in the specification and the set of `letting` statements in the instance file. A pair consisting of a valid specification and instance file has the same meaning as that of the specification with each `given` statement replaced by the corresponding `letting` statement in the instance file. Furthermore:

– for every `given` statement of the form "`given` *identifier*:*domain*" that appears in the specification, the instance file must contain exactly one

[5] This correspondence holds for unmerged statements, but merging is allowed.

statement of the form "`letting` *identifier* `be` *exp*" where *exp* evaluates to a member of *domain*,[6] and
- for every `given` statement of the form "`given` *identifier* `new type enum`" that appears in the specification, the instance file contains exactly one statement of the form "`letting` *identifier* `be new type enum` $\{NewID_1, \ldots, NewID_n\}$".
- The scope in which the statements in the instance file are made is defined as follows. Each `letting` statement is made in the scope of all the statements in the specification file that precede its corresponding `given` statement. That is, the constants and parameters already declared can be used in giving the value for this parameter.

To illustrate, consider Fig. 5, which is an instance file for the SONET specification given in Fig. 1. Notice that we distinguish an instance file from a specification file via the `language` statement.

A SONET Instance File

```
language  ESSENCE instance 1.2.0
letting   nrings be 2
letting   nnodes be 5
letting   capacity be 4
letting   demand be {{1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 5}}
```

Fig. 5. Example instance file for the SONET specification of Fig. 1

For an instantiation to be valid it must meet certain instantiation conditions. Explicit instantiation conditions are given by `where` statements, but a specification also imposes implicit instantiation conditions, such as:

- A declaration of the form "`letting` *id* `be new type of size` *e*:`int`" imposes the instantiation condition that $e \geq 1$.
- Use of the type "`matrix indexed by` $[\omega_1 \{ l_1..u_1 \}, \ldots, \omega_n \{ l_n..u_n \}]$" imposes the instantiation condition $l_i \leq u_i$ for all $1 \leq i \leq n$.
- Use of annotations of certain forms imposes the instantiation condition $e \geq 0$, for example: $\{$`size` *e*:`int` $\}$, $\{$`maxSize` *e*:`int` $\}$, $\{$`minSize` *e*:`int` $\}$, $\{$`partSize` *e*:`int` $\}$.

Observe that it may be possible that there are no instantiations that meet the instantiations conditions that are explicit and implicit in an ESSENCE specification. In such a case, the problem specification has no instances. In certain cases it may be possible for an implementation to detect this condition and issue a warning message.

[6] Observe that if *domain* is the empty domain, then the specification has no instances.

4 Abstraction in Essence

This section explains how the features of Essence, as introduced in the previous section, yield a language with a great deal of abstraction.

A guiding principle in the design of Essence has been that the language should not force a specification to provide unnecessary information or make unnecessary decisions. Existing languages often force a specification to introduce unnecessary objects or to unnecessarily distinguish between objects. This typically introduces symmetry into the specification. As will be demonstrated, the facilities that Essence has for abstraction enable this to be avoided.

The high level of abstraction provided by Essence is primarily a consequence of four features, which we discuss in the next four subsections.

4.1 Wide Range of Types

Essence supports a wide range of types and type constructors (including sets, multisets, tuples, relations, functions and partitions) and decision variables can have domains containing values of any one of these types. For example, the knapsack problem (i.e., Mystery Problem of Fig. 1) requires identifying the set of objects that are to go in the knapsack, so this is readily represented by a decision variable of type `set of` U, where U is the set of all objects. The objects themselves are a given enumerated type; there is no need to identify them with, say, integers. Notice that the specification of Garey and Johnson [14] does not identify the objects with integers and a specification language should not force one to do so.

4.2 Nested Types

Essence allows type constructors to be nested to arbitrary depth. This is the most important and distinctive feature of Essence, and could be considered our most important contribution to the design of constraint languages.

To observe its importance, consider the SONET problem, which 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 a set of nodes — and this can be stated directly and explicitly in Essence by using a decision variable of type set of sets.

Many languages support variables of type set of integer, but not nested sets. In such a language one would have to model the decision variable of SONET by a matrix of set variables, and the indices of this matrix would be symmetric. Thus, a limitation of the modelling language has forced the user to introduce a symmetry into the specification that is not present in the problem. Notice that neither this symmetry, nor anything corresponding to it, is present in the Essence specification of Fig. 1.

Alternatively, one could model the SONET problem with a decision variable whose type is a relation between the rings and the nodes. But, again,

this introduces a symmetry into the model as the individual rings are interchangeable. [7]

4.3 Quantification over Decision Variables

As we have seen, Essence allows quantifiers to range over values determined by a decision variable. For example, notice that the constraint of the GRP (see Fig. 1) is of the form $\forall \mathit{pair1}, \mathit{pair2} \subseteq \mathit{Ticks}.\mathit{Constraint}$, where *Ticks* is a decision variable of type set. Without this capability, the GRP constraint would need to take the form

$\forall \mathit{pair1}, \mathit{pair2} : \texttt{set}\ (\texttt{size}\ 2)\ \texttt{of int}\ (1..2 * *n).$

$(\mathit{pair1} \subseteq \mathit{Ticks} \wedge \mathit{pair2} \subseteq \mathit{Ticks}) \rightarrow \mathit{Constraint}$

Besides being awkward, a constraint of this form in any language leads to an implementation containing $\Theta(2^{4n})$ constraints, one for each way of drawing two pairs from a set of 2^n elements. In contrast, the Essence constraint that quantifies over the decision variable can be compiled to a model with $\Theta(n^4)$ constraints, one for each way of drawing two pairs from a set of n elements.

4.4 Unnamed Types

Essence provides types containing unnamed, indistinguishable elements, a feature necessary for adequate abstraction. Many problems involve some set of elements, yet do not mention particular elements. For example, we know of no specification or model of the SGP in which the constraints name any particular golfer; hence the golfers are indistinguishable. But since constraint languages do not have unnamed types, all models and specifications — other than the Essence specification in Fig. 1 — name the golfers either in the specification itself or in giving the input values. Naming, and hence distinguishing, the otherwise indistinguishable golfers introduces symmetry into the model, namely that the golfer names can be interchanged.

5 Implementing Essence

Although this paper is about the design of the Essence language, it is worth briefly considering our progress in and plans for using Essence as the input language to an automated modelling system, and providing citations to in-depth reports on this work.

We implement Essence using a two-stage process. The first stage is the *refinement* of an Essence specification into a constraint model in Essence′, a language derived from Essence mainly by removing facilities for abstraction and adding facilities common to existing constraint solvers and toolkits. The second stage is to *translate* an Essence′ model into the appropriate input for a particular constraint solver.

[7] This symmetry could be avoided by defining the rings to be an unnamed type, a facility discussed later.

To perform refinement, a rule-based system, called Conjure [11], is being developed. A prototype implementation of Conjure exists, which parses all of Essence 1.1.0 and refines specifications in a fragment of the language into model *kernels* in Essence′. We call these "kernels" because they do not contain many of the enhancements that characterise the most effective models, such as symmetry-breaking constraints and implied constraints. The kernels do, however, contain channelling constraints. Furthermore, as discussed in [12], an advantage of the refinement approach to formulating a model is that we can recognise when a modelling decision introduces a symmetry (as, for example, described in Sections 4.2 and 4.4) and build up a description of the symmetry present in the model produced. The kernels Conjure produces contain annotations describing this symmetry. There are difficulties in refining a language with arbitrarily-nested types and one of the principal contributions of [11] is to show how this can be done. Work is underway on a full implementation of Conjure: a front end for Essence 1.2.0 has been developed, and work on refinement is ongoing.

Essence′ can be thought of as a solver-independent modelling language and is somewhat similar to opl [33] or MiniZinc [27], both also solver-independent modelling languages. Given that it is intended to have a similar level of abstraction to existing constraint solvers, Essence′ supports only a limited number of types, and no nesting. It allows neither quantification over decision variables nor unnamed types. We have implemented a translator for mapping Essence′ to ECL^{i}PS^e [34] and another to map Essence′ to Minion [15, 16]. Translators to other solvers, such as Gecode[8], are planned.

The facilities offered by different constraint solvers vary, in terms both of the types of variables and the library of constraints supported. For example, some solvers support set variables, whereas others do not. Similarly, some global constraints, such as global cardinality [28], are not universally supported. If Essence′ contains a minimal set of such features, then the translation stage becomes more difficult for the more sophisticated solvers. Consider, for instance, the case where the target solver supports set variables, but Essence′ does not. In order to exploit the solver fully, the translator must reconstruct set variables from their representations as sets of constrained atomic variables in Essence′. This is difficult, and is likely simply to be reversing a refinement step performed by Conjure, which is wasteful. If, on the other hand, Essence′ contains a maximal set of such features, then the translation stage becomes more difficult for the least sophisticated solvers. In the opposite situation to the example above, where Essence′ supports set variables but the target solver does not, the translator must refine the set variable, hence replicating part of Conjure.

Our solution to this dilemma is that we plan to support *variants* of Essence′ to support common combinations of variable and constraint types. The correct variant would be chosen according to the intended target solver. This allows us to make full use of the facilities of the target solver, while not forcing a translator to perform refinement.

[8] www.gecode.org

6 An Evaluation of the Use of Essence

To evaluate Essence we specified a large suite of problems in the language and here reflect on the process and results. A suite of 133 problems, both theoretical and practical, was selected, drawn from CSPLib and the literature. Specification was undertaken by two undergraduates in computer science, neither with any previous experience of constraint programming. Each student was easily able to adapt to Essence by drawing on his understanding of discrete mathematics.

Specification began by obtaining an unambiguous natural language description of the problem. The flexibility of Essence allowed specifications to be written directly from this description. The key decision concerned the representation of the decision variables; from this the constraints followed easily. Attention had to be paid to abstraction in order to make full use of the language. Some of the problems were described in the literature in terms of low-level objects such as matrices. With the goal of producing an abstract specification there was typically a single obvious choice for the type of the decision variable. This was not, however, always the case: in the SONET problem the configuration can be represented as a relation from rings to nodes or as a set of sets of nodes. The latter is preferable as it avoids having to name the rings.

Specification grew easier with experience; many specifications contained reusable common idioms. Another advantage of Essence is that similarities are present in abstract specifications that would not necessarily appear in the concrete constraint programs due to differing modelling choices.

The resulting catalogue [35] contains Essence specifications for the problem suite and, for comparison, previously-published specifications in Z, esra, opl and $\mathcal{F}$. The relative expressiveness and elegance of Essence is clearly demonstrated. Throughout, the specification length is proportional to the size of the problem statement, with larger examples being just as easy to read.

7 Comparison with Other Languages

Constraint modelling languages have their roots in algebraic modelling languages, dating from the 1970s and originating in the field of mathematical programming (e.g. GAMS [2] and MGG [31]). These languages made three significant advances. First, they were developed to simplify the user's role in solving mathematical programming problems, providing a syntax much closer to problem descriptions found in the literature. Second, they were 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 separated 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 [7]).

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 [21] 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 [4], CHIP [1]) or C++ (Ilog Solver [19]) 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 programming language, and interleave the specification of a problem with how it is solved. More recently, in parallel with an increasing awareness of the constraint modelling bottleneck, declarative constraint modelling has re-emerged in languages such as EaCL [25], OPL [33], ESRA [6], $\mathcal{F}$ [18], and NP-Spec [3].

In both declarative and procedural constraint languages, the trend has been towards an increase in abstraction. For example, ECL^{i}PS^e supports decision variables whose domain elements are sets [17]; similarly $\mathcal{F}$ supports functions, ESRA supports relations and functions, and NP-Spec supports sets, permutations, partitions and integer functions. Each increase in abstraction allows the user to ignore additional modelling decisions, leaving this to the compiler. ESSENCE made a large leap in this direction by providing a range of types wider than previous languages and introducing type constructors that can be nested arbitrarily. The magnitude and importance of this leap cannot be over-emphasised; it is largely responsible for making ESSENCE a problem specification language rather than a modelling language.

To realise fully the abstraction provided by a rich type system a language must permit quantification over decision variables, rather than just fixed ranges of integers. Other than ESSENCE we know of only three constraint languages that provide this feature: ESRA, $\mathcal{F}$ and LOCALIZER [24]. It is worth noting that these are all declarative languages. In contrast, programming languages that are augmented with constraint facilities, such as Solver and ECL^{i}PS^e, do not have direct support for quantification. Instead they provide iteration, which allows a direct implementation of quantification over a fixed set; however attempting to use iteration to provide quantification over a variable expression tends to lead to the problem raised in Section 4.3.

Further abstraction is obtained through the provision of unnamed types. As far as we know, ESSENCE is the only constraint language to provide such a facility.[9]

The constraint language closest to ESSENCE in terms of features is the recently-developed Zinc language [22]. There are two principal criteria by which Zinc and ESSENCE may be distinguished. The first is extensibility: Zinc is extensible via user-defined functions and predicates, a feature which ESSENCE lacks. The second is in the degree of abstraction of, and quantification over, decision variables. The set of atomic types provided by Zinc and ESSENCE are similar, although Zinc supports floats, which ESSENCE does not, and unnamed types are unique to ESSENCE, as noted above. Both languages support type

[9] Frisch and Miguel [13] argue that the sets of unnamed objects provided by ESRA are a different facility that does not yield all the benefits of unnamed types.

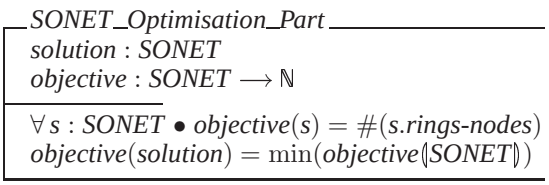

Fig. 6. Part of a Z specification of the SONET problem.

constructors that can be nested to arbitrary depth. The two languages have several type constructors in common, such as sets, arrays and tuples, and some unique to one of the languages, such as records and lists (Zinc). Through its multiset, partition, function, and relation type constructors, Essence supports more abstract decision variables than Zinc, which can only replicate variables of this kind through a constrained collection of variables of more primitive type - i.e. through modelling. Essence's support for quantification over decision variables adds significantly to its expressive power and is vital for concision when dealing with variables with nested domains.

Another approach to problem specification is to employ a more-powerful, more-general specification language such as Z. This approach has been explored thoroughly by Renker and Ahriz [29], who have built a toolkit of Z schemas to support common global constraints and other common idioms and have used this to build a large catalogue of specifications [30]. A shortcoming of this approach is that Z is too general for the task as it allows specifications of problems that do not naturally reduce to CSPs. For example, unlike Essence, nothing prevents a decision variable from having no domain or an infinite domain and nothing prevents the use of a decision variable to specify the size of a matrix of decision variables. Of course, one could try to identify a subset of Z that is suitable for the task, but we believe doing this and enhancing the language with a suitable schema library would result in a language that approximates Essence. Furthermore, an inherent limitation of Z is that it provides no mechanism for distinguishing parameters from decision variables, a distinction that is central to the notion of a problem.

A further shortcoming of specifications in a language like Z is that they are far less natural than those in Essence. To observe this, compare the equivalent specifications, available at [30], of the SONET problem in the two languages. Fig. 6 shows part of that Z specification, which is equivalent to the Essence statement "`minimising` |*rings-nodes*|".

The Alloy language [20] avoids some of these shortcomings of Z by restricting itself to first-order logic. Alloy gives a natural and expressive way of specifying problems, in particular model checking problems, in terms of relations and atoms, and maps these specifications to efficient SAT models. The main drawbacks to using Alloy as a constraint modelling language are the limited type system and the lack of explicit declarations of variables — common to model checking — which makes the mapping of Alloy specifications to efficient constraint models difficult.

In order to illustrate the advances made by Essence, we compare the specifications of the SGP in Essence to four other languages: Zinc, NP-Spec, esra and opl. These languages are representative of a range of existing constraint languages; opl is similar to both MiniZinc (a simplified version of Zinc) and Essence′, and esra is similar to $\mathcal{F}$. Both an English definition and an Essence specification of this problem are given in Fig. 1. Specifications in Zinc (from [23], but with symmetry-breaking constraints omitted for brevity) esra (from [6]), opl (written by the authors) and NP-Spec[10] are given in Fig. 7. Each specification is divided into four sections. The first declares any parameters, types and constants (and in the case of Zinc, declares a predicate), the second declares the variables, the third imposes that the variables represent a set 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 set (as the weeks are not distinguished) of partitions of the players, where each partition gives the groups 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 a set of partitions and so it cannot be used. Each of the Zinc, NP-Spec, opl and esra specifications therefore uses a specific implementation of sets of partitions for the variables, requiring the user to make modelling decisions. The NP-Spec and esra specifications both implement the set of partitions by introducing a range of values which represent which group a player plays in, and then use a total 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 two-dimensional matrix indexed by players and weeks, with entries drawn from a range to denote the group. The Zinc specification takes a slightly different approach, using a two-dimensional matrix indexed by weeks and groups, with entries that are sets of players. Given Zinc's ability to support nested types, this specification could have been written more concisely, for example as a single-dimensional matrix indexed by weeks, with entries that are sets of sets of players. Zinc does, however, lack the partition type constructor needed to match the Essence specification.

Given the decision variables they employ, each of the Zinc, esra, opl and NP-Spec specifications have to constrain the decision variables to represent a set of partitions. These constraints are contained in the third section of each specification.

As they label the players, weeks and groups, the Zinc, esra, opl and NP-Spec specifications introduce a large amount of symmetry that is not present in the original specification. In Essence, 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 dependency between the groups in each different week 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

[10] From `http://www.dis.uniroma1.it/~cadoli /research/projects/NP-SPEC/`

Zinc

```
1a  int: Weeks; int: GroupSize; enum Players = ...;
1b  int: Groups = card(Players) div GroupSize;
1c  assert Groups * GroupSize == card(Players) : "invalid number of players";
1d  predicate maxOverlap(list of var set of $E: sets, int: n) =
        forall(i,j in 1..length(sets) where i < j)
            (card(sets[i] intersect sets[j]) =< n);
2   array[1..Weeks,1..Groups] of var set of Players: group;
3a  constraint
        forall (i in 1..Weeks)
            (maxOverlap([group[i,j] | j in Groups], 0)) ;
3b  constraint
        forall (i in 1..Weeks, j in 1..Groups)
            (card(group[i,j]) == GroupSize);
4   constraint
        maxOverlap([group[i,j] | i in 1 .. Weeks, j in Groups], 1);
```

ESRA

```
1a  cst weeks, groups, groupsize : N
1b  dom Players = 1..groups * groupsize,
    Weeks = 1..weeks, Groups = 1..groups;
2   var Sched : (Players × Weeks) →^(groupsize×weeks) Groups
    solve
3       ∀(h : Groups, w : Weeks)count(groupsize)
            (p : Players|Sched(p, w) = h)      ∧
4       ∀(p1 < p2 : Players)count(0...1)
            (w : Weeks|Sched(p1, w) = Sched(p2, w))
```

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 {
3       forall(w in Weeks & g in Groups) (sum(p in Players)
            (Schedule[p,w] = g) = groupsize);
4       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. 7. Zinc, ESRA,OPL and NP-SPEC specifications of the Social Golfers Problem.

groups, the Essence specification avoids this problem. The full extent of the symmetries of the groups has been missed previously (for example [5]), which shows how the use of Essence and Conjure can aid even expert modellers.

One unusual feature in esra's definition of "Sched", on line 2, is the notation $\rightarrow^{groupsize*weeks}$. 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 set 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 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, as per the second Essence specification in Fig. 1. 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 second Essence specification, using the common transformation that instead of imposing that 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. The Zinc specification imposes this constraint in a similar way to the first Essence specification in Fig. 1. It constructs a list of all groups of players and employs a user-defined predicate to ensure that the intersection of each pair of groups has cardinality at most one. Note that the Zinc constraint in this specification contains some redundancy: it is not necessary to compare groups in the same week given the constraint representing the partitioning of the players.

8 Conclusions

In appraising Essence 1.2.0, we have identified two main shortcomings, which we discuss below. There are also a number of features that were not added to this version of the language, but which are ongoing work; these are also discussed, at the end of this section. One issue that this paper does not address is the expressive power of Essence. However, this topic has been considered carefully by Mitchell and Ternovska [26].

As noted in the introduction, a semantics of an earlier version of Essence is given in [8]. This semantics works on instance specifications, i.e. where values have been given for parameters. It specifies a denotation function that maps a specification and an assignment of values to its decision variables to a truth value. The denotation function is not total; the denotation of a specification relative to an assignment may be undefined. The rationale for this is that it is necessary to handle cases where, for example, an array index is out of bounds or a divisor is zero. It is tempting to assign such expressions "false", but this is problematic. To illustrate, consider the constraint C_1: $X/Y = Z$. Assuming that the reader agrees that C_1 is not satisfied by an assignment that maps Y to 0, this constraint is equivalent to the conjunction: $(Y \neq 0) \wedge (X = Y * Z)$. The

difficulty arises when C_1 is used within a logical expression, as in: $\neg(C_1)$. Is this equivalent to $(Y \neq 0) \wedge (X \neq Y * Z)$, or perhaps $\neg((Y \neq 0) \wedge (X = Y * Z))$, or perhaps something else? One approach is to remove division (and other sources of undefined expressions) from the language, thus forcing the user to choose between these options. The question is whether the restrictions this places on the language are acceptable.

The reader may wonder why it is necessary to cater for undefined expressions at all. Perhaps a specification that contains an undefined expression should be invalid. However, Essence is sufficiently rich that undefinedness cannot be detected or eliminated statically. Hence, we have a situation where some assignments may be undefined, whereas others are valid solutions, and we wish to be able to reach these valid solutions.

Supporting undefined expressions, as in the scheme of [8], has its own share of difficulties, not least the difficulty of their implementation. Typically, constraint languages that support complex constraints do so by rewriting them into a conjunction of simple constraints. Hence, it is important to have a semantics according to which these rewriting rules preserve equivalence. A simple example is rewriting $u - u = 0$ to $0 = 0$, if u is an undefined expression. A second example demonstrates a shortcoming of the semantics presented in [8]: consider the common step of rewriting $exp_1 \vee exp_2$ to $(b_1 = exp_1) \wedge (b_2 = exp_2) \wedge (b_1 \vee b_2)$. According to this semantics, this transformation does not preserve equivalence. To see this, consider the case where exp_1 is undefined, and exp_2 is true: now $exp_1 \vee exp_2$ is true, but the first conjunct of the rewritten expression is false, hence the whole is false. In future work, we plan to revisit the semantics of Essence to address these issues and find an acceptable semantics that is also efficient to implement.

The second major shortcoming is in our handling of matrices. When designing Essence, we felt that it was important to allow matrix indices to depend on the parameters of a particular problem instance. We also wanted to be able to perform type checking at the translation stage (of the TIS schema). As a result, we decided that matrix indices should not be part of the type of a matrix. However, this conflicts with our design decision not to allow any collection type to contain matrices with different indices. An obvious remedy is to extend the type of a matrix to include its indices. This leaves the problem of how to check for well-typedness at the translation stage, when the indices may not be known until the instantiation stage. Our present solution to this is to turn any conditions that cannot be completely checked at the translation stage into implicit "where" statements that apply restrictions to the combinations of parameters that are acceptable, i.e. such that checking is not necessary at the solving stage. To illustrate, consider a specification that includes the constraint $M_1 \leq_{\text{lex}} M_2$. This generates the implicit where statement that the smallest index value of M_1 is equal to the smallest index value of M_2. A general solution, which we plan to investigate further, is to embrace this approach fully: consider matrix index values part of the type, and similarly augment other types as necessary. Then, where the type checker cannot determine if a type condition is met (due to the presence of parameters), generate `where` statements that enforce the type condition during instantiation.

As well as addressing the above shortcomings, we have identified a number of other important directions for future work. We plan to continue the development of Essence by specifying a much wider range of problems in the language, and using this to guide the design of further features and enhancements. This will inevitably lead to the incorporation of additional operators for constructing expressions and additional type constructors, such as lists, trees and graphs. We also plan to explore the addition of set and matrix comprehensions. To incorporate standard quantifiers, such as product, union, intersection, min and max, we will develop a general mechanism for using any associative and commutative operator as a quantifier. Following Zinc, we are considering the addition of features for modularity and extensibility, which would promote further concision and modularity in Essence specifications.

In summary, Essence is a formal language that is natural in that it is accessible to someone with an understanding of discrete mathematics but not constraint programming. The central contribution of Essence is the introduction of 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. This, coupled with the ability to quantify over decision variables, is fundamental to how 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.

Acknowledgements

We thank Andy Grayland, David Mitchell, Andrea Rendl and Eugenia Ternovska for useful discussions about Essence, and Matt Grum and Rupert Thompson for writing the catalogue of Essence specifications. Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship. Warwick Harvey is supported by UK EPSRC grant EP/D030145/1.

References

1. A. Aggoun and N. Beldiceanu. Overview of the CHIP compiler system. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 421–436. MIT Press, London, 1993.
2. A. Brooke, D. Kendrick, and A. Meeraus. *GAMS: A Users' Guide*. The Scientific Press, Danvers, Massachusetts, 1988.
3. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, 26:165–195, 2000.
4. A.M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M.G. Wallace. ECLiPSe: An introduction. Technical Report IC-Parc-03-1, Imperial College London, 2003.
5. P. Flener, A.M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, and T. Walsh. Matrix modelling. In *Proceedings of the CP'01 Workshop on Modelling and Problem Formulation*, pages 1–7, 2001.

6. P. Flener, J. Pearson, and M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Proceedings of LOPSTR '03: Revised Selected Papers*, volume 3018 of *Lecture Notes in Computer Science*, 2004.
7. R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Thomson/Brooks/Cole, Pacific Grove, California, second edition, 2003.
8. A. M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The essence of ESSENCE: A language for specifying combinatorial problems. In *Proc. of the 4th Int. Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 73–88, 2005.
9. A. M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of ESSENCE: A language for specifying combinatorial problems. In *Proc. of the Twentieth Int. Joint Conf. on Artificial Intelligence*, 2007.
10. A. M. Frisch, B. Hnich, I. Miguel, B. M. Smith, and T. Walsh. Transforming and refining abstract constraint specifications. In *Proceedings of the Sixth Symposium on Abstraction, Reformulation and Approximation*, volume 3607 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2005.
11. A. M. Frisch, C. Jefferson, B. Martínez Hernández, and I. Miguel. The rules of constraint modelling. In *Proc. of the Nineteenth Int. Joint Conf. on Artificial Intelligence*, pages 109–116, 2005.
12. A. M. Frisch, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. Symmetry in the generation of constraint models. In *Proceedings of the International Symmetry Conference*, 2007.
13. A. M. Frisch and I. Miguel. The concept and provenance of unnamed, indistinguishable types. Available at www.cs.york.ac.uk/aig/constraints/AutoModel/, September 2006.
14. M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.
15. I. P. Gent, C. Jefferson, and I. Miguel. Minion: Lean, fast constraint solving. In *Proceedings of the 17th European Conference on Artifical Intelligence*, 2006.
16. I. P. Gent, I. Miguel, and A. Rendl. Tailoring solver-independent constraint models: A case study with essence' and minion. In *Proceedings of the 7th International Symposium on Abstraction, Reformulation and Approximation*, pages 18–21, 2007.
17. C. Gervet. Conjunto: Constraint logic programming with finite set domains. In Maurice Bruynooghe, editor, *Logic Programming — Proc. of the 1994 International Symposium*, pages 339–358. The MIT Press, 1994.
18. B. Hnich. *Function Variables for Constraint Programming*. PhD thesis, Computer Science Division, Dept. of Information Science, Uppsala University, 2003.
19. *ILOG Solver 6.3 User Manual*. ILOG, S.A., Gentilly, France, July 2006.
20. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
21. J-L. Lauriere. ALICE: A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.
22. K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. De la Banda, and M. Wallace. The design of the zinc modelling language. *Constraints*, This Volume, 2008.
23. K. Marriott, R. Rafeh, M. Wallace, M. Garcia de la Banda, and N. Nethercote. Zinc 0.1: Language and libraries. Technical report, Monash University, 2006.
24. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5(1/2):43–84, 2000.

25. P. Mills, E.P.K. Tsang, R. Williams, J. Ford, and J. Borrett. EaCL 1.5: An easy abstract constraint optimisation programming language. Technical report, University of Essex, Colchester, UK, December 1999.
26. D. Mitchell and E. Ternovska. Expressive power and abstraction in essence. *Constraints*, This Volume, 2008.
27. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543, 2007.
28. J-C. Regin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th Conference on Artificial Intelligence*, pages 209–215, 1996.
29. G. Renker and H. Ahriz. Building models through formal specification. In *Proc of the First Int. Conf. on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3011 of *Lecture Notes in Computer Science*, pages 395–401. Springer, 2004.
30. www.comp.rgu.ac.uk/staff/ha/ZCSP/, 2006.
31. R.V. Simons. Mathematical programming modeling using MGG. *IMA Journal of Mathematics in Management*, 1:267–276, 1987.
32. J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, 1989.
33. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
34. M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.
35. www.cs.york.ac.uk/aig/constraints/AutoModel, 2007.