The Design of ESSENCE: A Constraint Language for Specifying Combinatorial Problems

Alan M. Frisch[1] **Matthew Grum**[1] **Chris Jefferson**[2]
Bernadette Martínez Hernández[1] **Ian Miguel**[3]

[1]Artificial Intelligence Group, Dept. of Computer Science, Univ. of York, UK. {frisch, grum, berna}@cs.york.ac.uk
[2]Oxford University Computing Laboratory, Univ. of Oxford, UK. chrisj@comlab.ox.ac.uk
[3]School of Computer Science, Univ. of St. Andrews, UK. ianm@dcs.st-and.ac.uk

Abstract

ESSENCE is a new 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, thus 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 directly specified by using a decision variable whose type is precisely that combinatorial object.

1 Introduction

This paper describes ESSENCE,[1] a new language for specifying combinatorial (decision or optimisation) problems at a high level of abstraction. ESSENCE is the result of our attempt to design a formal language that enables 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 [1979].

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 [Spivey, 1989] or NP-SPEC [Cadoli *et al.*, 2000]. This has resulted in a language that, to a first approximation, is a constraint language, such as OPL [Van Hentenryck, 1999], $\mathcal{F}$ [Hnich, 2003] or ESRA [Flener *et al.*, 2004], enhanced with features that greatly increase its level of abstraction. Most importantly, as a combinatorial problem requires finding a certain type of combinatorial object, ESSENCE provides decision variables whose domain elements are combinatorial objects of that type and constraints that operate on that type. This enables problems to be stated directly and naturally; without the decision variables of the appropriate type the problem would have to be "modelled" by encoding the desired combinatorial object as a collection of constrained decision variables of some other type, such as integers.

As our motivations — and hence our methodology and results — for developing ESSENCE differ greatly from those that have lead 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 constraint modelling. Constraint modelling is the process of reducing a given problem to the 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. We call such variables atomic and atomic set variables, respectively. As an example, consider the Social Golfers Problem (SGP), which requires partitioning a set of golfers into equal-sized groups in each week of a tournament subject to a certain constraint. Thus, the goal is to find a multiset of regular partitions[2] that satisfies the constraint. A model for the SGP must therefore represent this multiset of regular partitions by a collection of constrained atomic or atomic set variables. There are at least 72 ways that this can be done [Frisch *et al.*, 2005c].

To automatically generate models for a problem, one must start with a formal specification of the problem and this specification must be 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 satis-

[1]ESSENCE version 1.1.0 is used within this paper. A full specification of the language can be found at [York Web, 2006].

[2]A partition is *regular* if all its subsets have the same cardinality.

fying 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 insisted on describing 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 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. This paper does not present a formal syntax (this can be found at [York Web, 2006]) or a formal semantics (this can be found in [Frisch *et al.*, 2005a]).

Mystery Problem:

`given` U `enum(...)`, $s : U \rightarrow$ `int` $(1...)$,
$v : U \rightarrow$ `int` $(1...)$, B, K: `int`
`find` U': `set of` U
`such that` $\sum_{u \in U'} .s(u) \leq B,\ \sum_{u \in U'} .v(u) \geq K$

Golomb Ruler Problem (GRP): Given n, put n integer ticks on a ruler of size m such that all inter-tick distances are unique. Minimise m.

`given` n : `int`
`where` $n \geq 0$
`letting` $bound$ `be` 2^n
`find` $Ticks$: `set (size` n`) of int` $(0..bound)$
`minimising` $\max(Ticks)$
`such that` $\forall_{pair1, pair2 : \texttt{set (size 2) of int} \subseteq Ticks} . pair1 \neq pair2 \implies$
$\max(pair1) - \min(pair1) \neq \max(pair2) - \min(pair2)$

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 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.* [2005b])

`given` *nrings*, *nnodes*, *capacity* : `int` (1...)
`letting` *Nodes* `be int`(1..*nnodes*)
`given` *demand* : `set of set (size` 2`) of` *Nodes*
`find` *network* : `mset (size` *nrings*`) of`
`set (maxsize` *capacity*`) of` *Nodes*
`minimising` $\sum_{ring \in network} |ring|$
`such that` $\forall_{pair \in demand} . \exists_{ring \in network} . pair \subseteq ring$

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.)

`given` w, g, s : `int` (1...)
`letting` *golfers* `be new type of size` $g * s$
`find` *sched* : `mset (size` w`) of rpartition (size` s`) of` *golfers*
`such that` $\forall_{week1, week2 \in sched} . week1 \neq week2 \implies$
$\forall_{group1 \in week1, group2 \in week2} . |group1 \cap group2| < 2$
Alternative constraint:
`such that` $\forall_{golfer1, golfer2 : golfers} . golfer1 \neq golfer2 \implies$
$(\sum_{week \in sched} .\texttt{together}(golfer1, golfer2, week)) < 2$

Figure 1: ESSENCE specifications of four problems.

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?

You should have been able to identify this as the Knapsack decision problem because the ESSENCE specification is nearly identical to the specification given by Garey and Johnson [1979, 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 as 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 to the problem), what combinatorial objects must be found (the decision variables) and what constraints the objects must be 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 by the keywords `given`, `where`, `letting`, `find`, `maximising`, `minimising` and `such that`. `letting` statements declare constant 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. `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.

An ESSENCE specification is a list of statements composed according to the regular expression:

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

Now consider the specification of the Golomb Ruler Problem (GRP, problem 6 in CSPLib) in Fig. 1, and the instance obtained by letting n be 4. The domain of the decision variable $Ticks$ contains all sets of four elements drawn from $\{0, .., 16\}$. Consider an assignment of $Ticks$ 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, $\max(\textit{pair1}) - \min(\textit{pair1})$ is different from the distance between the other $\max(\textit{pair2}) - \min(\textit{pair2})$. We believe that the reader will concur that the ESSENCE specification closely matches the given English description, and is substantially closer than a standard CSP model of the problem.

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

Constraints are built from parameters, constants, quantified variables and decision variables using operators commonly found in mathematics. ESSENCE also includes variable binders such as $\forall_x$, $\exists_x$ and Σ_x, where x can range over any specified finite domain (e.g. integer range but not integer). The GRP constraint can be 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. The decision variable is represented straightforwardly as a multiset (the fact that it is a set is an implied constraint) of regular partitions, (regularity guarantees equal-sized 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 elements 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 the pairs of golfers, ensuring that they are partitioned together (via the global constraint `together`) over the weeks of the schedule at most once. Note that here we make use of a facility common to constraint languages: treating Booleans as 0/1 for the purpose of counting.

3 The Features of ESSENCE

This section explains the most significant features of ESSENCE. The first two subsections discuss types and domains and the next two discuss expressions and quantification.

ESSENCE is a strongly-typed language; every expression has a type and the parser that can infer the types of expressions and can perform type-checking. 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 arguments.

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 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 are sets that contain all elements that have a similar structure, whereas domains are 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.

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 comes from the fact that typing is static and that all types are disjoint.

The atomic types of ESSENCE are `int` (integer), `bool` (Boolean), user-defined enumerated types and user-defined unnamed types. The two user-defined types are defined through letting or given statements such as the following:

```
letting players be new type enum {alan, ian, chris, berna}
letting rings be new type of size 4
given players new type enum (...)
```

The first statement defines a new type comprising four named atomic elements. As all types are disjoint, the elements of this type cannot be members of any other type, and the four names cannot be used to name anything else. The second statement also defines a new type comprising four elements; however these elements are not named. These four elements are distinct from the elements of all other types. The third statement is similar to the first except that the enumeration of the elements of the type is provided as input rather than as part of the specification.

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 $F < T$, 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. If $\tau, \tau_1, \tau_2, \ldots$ are the names of any types, $\theta_1, \theta_2, \ldots$ are the names of any ordered types, ϕ is the name of a finite type (`bool`, an enumerated type or an unnamed type) and n is any positive integer then the following all name compound types:

`set of` τ	(a finite set drawn from τ)
`mset of` τ	(a finite multiset drawn from τ)
$\tau_1 \rightarrow \tau_2$	(a finite[3] partial function with domain τ_1 and codomain τ_2)
`tuple` $\langle \tau_1, \ldots, \tau_n \rangle$	(an n-tuple)
`rel` $\tau_1 \times \cdots \times \tau_n$	(a finite n-ary relation)
`partition of` ϕ	(a partition of the elements of ϕ)
`rpartition of` ϕ	(a regular partition of the elements of ϕ)
`matrix indexed by` [$\theta_1, \ldots, \theta_n$] `of` τ	(an n-dimensional matrix)

For example, `set of int` and `rel int × 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
rel partition of players × mset of set of int
```

Notice that finiteness 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.

Finally notice that the parameters and decision variables of a specification cannot enter the type names. This restriction is necessary to enable the type of every expression in a specification to be determined before the parameters are instantiated with values. This is why we say that ESSENCE is statically typed.

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 parenthesis, the type underlying a domain can always be obtained by removing the parenthesised subexpressions. Thus, the types underlying the above two domains are `int` and `set of int`, respectively.

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 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)
`int` (1, 3, 4..10)	(a mixture of values and ranges)
`int` (1..)	(the positive integers)
`int` (..-1,1..)	(the non-zero integers)
`int` ($l..u$)	(the integers from l to u inclusive)
`int` ($S \cup \{0, 1\}$)	(the integers in S and 0 and 1)

In these last two examples, l, u and S can be parameters or identifiers declared by a `letting` statement. This illustrates another important distinction between domains and types. Parameters can appear in the annotations of a domain but they cannot appear in a type. However, decision variables cannot appear in domains, a requirement that is needed if specifications are to be mapped to CSPs.

Now consider how type constructors can be annotated. The set, multiset, partition, regular partition and relation constructors can all be annotated by inserting, before the keyword "`of`," a size restriction of the form (`size` *intexp*) or (`maxsize` *intexp*), where *intexp* is an integer expression that contains no decision variables. Thus, all the following are domains:

```
set (size n) of int
mset (maxsize n+2) of int (1..100)
partition (size m) of mset (maxsize 4) of int (1..100)
```

The second and third of these domains 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.

In addition to the size annotation, the relation constructor can also be annotated with *multiplicities* that specify, for example, that a relation is n-to-m. As we have modelled this feature on a similar feature provided by ESRA, we will not describe it further.

The function 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:

```
players →(total) players
players →(injective) players
players →(total surjective) players
```

The matrix and tuple type constructors take no annotations, though, of course, the types nested in them can be annotated, as in these examples:

```
tuple ⟨ int (1..), set (size 2) of int ⟩
matrix indexed by [players, int (5..10)] of bool
```

[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.

In all uses of the matrix constructor the domains for the indices must be a single range of an ordered type, as illustrated in the second statement above.

Every domain is either finite or infinite and the domains associated with a decision variable must be finite, which is required to ensure that specifications map to CSPs. Elsewhere [Frisch *et al.*, 2005a] we have given a set of rules for generating all finite domains. This paper instead relies on the reader's intuition and understanding to determine if a domain contains a finite or infinite set of elements.

Expressions The expressions of ESSENCE are formed in a manner much as one would expect, bearing in mind that every expression has a type. Constraints and the restrictions in `where` statements are expressions of type Boolean and objective functions are expressions of any ordered type.

The atomic expressions of ESSENCE are the constants, parameters, decision variables and quantified variables. Other than the unnamed types, and compound types constructed from them, every value has a name. These names are the constants of the language.

We shall not explain the syntax for naming constants, but note that our objective was to give every value a distinct name. A difficulty is that the emptyset is a member of every type of the form `set of` τ. Our remedy is to explicitly attach the type to every emptyset. Thus, for example $\{\}$:`set of int` and $\{\}$:`set of` *players* are distinct names for distinct objects.

Compound expressions are formed by applying operators to expressions. ESSENCE provides a wide range of operators drawn from discrete mathematics (e.g., intersection, function application, projection, set membership), logic (e.g., conjunction, implication), and the global constraints found in constraint programming (e.g., alldifferent, global cardinality, lexicographic ordering). Whereas the global constraints of other constraint languages apply only to matrices of integers, those of ESSENCE apply to matrices of any reasonable type. For example, in ESSENCE, alldifferent applies to a matrix of any type and lexicographic ordering applies to two one-dimensional matrices of any ordered type.

The type of a compound expression is a function of its operator and the *types* of its operands. 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.

Quantification 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 is in the "`minimising`" statement of the SONET specification. Each quantification expression consists of three components: a quantifier, either $\sum$, $\forall$ or $\exists$; followed by a non-empty list of variables (these are the quantified variables that are being "declared"); followed by a binding expression that dictates a finite set of values over which the variables range. These values are all of the same type, and this is taken as 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.

Binding expressions employ two methods of dictating these values. The first is to give a *finite* domain, as in the the universal quantifier of the alternative constraint of the SGP specification in Fig. 1. The second method is to obtain the values by taking all elements or (strict or non-strict) subsets of a set, partition or regular partition. All the quantification expressions in the Knapsack and SONET specifications in Fig. 1 draw their values from a set. The set, partition or regular partition can be denoted by an arbitrary expression, including expressions that contain decision variables, as seen in the examples mentioned. In quantification expressions of this kind there is no need to specify the types of the quantified variables as they can easily be inferred. For example, in the quantified expression $\sum_{u \in U'}$ of the Knapsack specification it is clear that u must be of type U since U' is of type `set of` U. Finally it is also possible to employ both methods to dictate the values taken by the quantified variables. 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.[4]

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 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.

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 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 [1979] does not identify the objects with integers and a specification language should not force one to do so.

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

[4]In this case the domain is not finite, but finiteness is obtained since the sets must be a subset of $Ticks$, which is finite.

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.

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 $\forall_{pair1, pair2 \subseteq Ticks} . Constraint$, where *Ticks* is a decision variable of type set. Without this capability, the GRP constraint would need to take the form

$$\forall_{pair1, pair2 : \texttt{set (size } 2\texttt{) of int } (1..2^n)} \cdot \\ (pair1 \subseteq Ticks \wedge pair2 \subseteq Ticks) \Longrightarrow Constraint$$

Besides being awkward, this always 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.

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 the golfer names can be interchanged.

5 Implementing ESSENCE

At this point we have implemented in Haskell a parser for all of ESSENCE 1.1.0; it performs complete syntactic analysis, including all necessary type checking and type inference. A second implementation of this same parser in Java is nearing completion.

We have implemented a rule-based system, called CONJURE [Frisch *et al.*, 2005c], that can translate —we say refine — specifications in a fragment of ESSENCE into model *kernels* at a level of abstraction supported by existing modelling languages. We call these "kernels" because they do not contain many of the enhancements that characterize the most effective models, such as symmetry-breaking constraints and implied constraints.

The model kernels generated by CONJURE are expressed in ESSENCE′, a subset of ESSENCE that has a level of abstraction that is supported by existing constraint solvers. As such, ESSENCE′ only supports atomic variables, atomic set variables, and matrices of these. It allows neither quantification over decision variables nor unnamed types. Thus, ESSENCE′ can be thought of as a solver-independent modelling language and is somewhat similar to OPL, another solver-independent modelling language.

Because ESSENCE′ has a level of abstraction similar to existing solvers it is not extremely difficult to translate ESSENCE′ models into existing languages. In particular, the translation can be performed without making any modelling decisions. We have implemented a translator for mapping ESSENCE′ to Eclipse [Wallace *et al.*, 1997] and are currently developing another one to map ESSENCE′ to Minion [Gent *et al.*, 2006].

One reason why new types of decision variables have been incorporated into constraint programming languages so slowly has been the difficulty of implementing the enhancements. Frisch *et.al.* [2005c] identify a difficult, fundamental problem in refining types that can be nested to arbitrary depth and they present a solution to it. With this breakthrough, and other techniques pioneered in the development of the CONJURE prototype, we believe we have all the technology needed to refine all of ESSENCE to kernel models.

Of course, our ultimate goal is to develop a refinement system that can refine any ESSENCE specification into a single, complete, effective ESSENCE′ model. We consider this to be the goal of achieving fully-automated modelling, something which we do not consider achievable in less than a lifetime. As explained earlier, the development of a suitable problem specification language is necessary to embark on the venture.

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 58 problems, both theoretical and practical, was selected, 26 drawn from CSPLib, and 32 from the literature. Specification was undertaken by an undergraduate in computer science with no previous experience of constraint or logic programming. He 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 [York Web, 2006] 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

Algebraic modelling languages, dating from the 1970s and originating in the field of mathematical programming (e.g. GAMS [Brooke *et al.*, 1988] and MGG [Simons, 1987]), made two significant advances. Firstly they were developed to simplify the user's role in solving mathematical programming problems, providing a syntax much closer to the expression of these problems found in the literature. Secondly, they are declarative, characterising the solutions to a problem rather than how the solutions are to be found. This lifts a burden from the user, and allows different solvers to be easily applied to the same problem. Many other useful modelling languages have been created in other areas, for example ACE [Fuchs and Schwitter, 1996], an English-like modelling language for reasoning about knowledge bases.

The success of modelling languages suggests that a similar approach might be fruitful for constraint solving. From early in the development of the field, the ALICE language [Lauriere, 1978] shares many features with algebraic modelling languages of the time, including its declarative nature.

Constraint programming languages have gradually evolved a greater range of types for decision variables. For example, Eclipse [Gervet, 1994] supports decision variables whose domain elements are sets; 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 makes a large leap in this direction by providing a range of types wider than previous languages and, uniquely, type constructors that can be arbitrarily-nested. The magnitude and importance of this leap cannot be over-emphasized; it is largely responsible for making ESSENCE a problem specification language rather than a modelling language. Zinc [Marriott *et al.*, 2006; de la Banda *et al.*, 2006] is a new specification language that follows ESSENCE in providing arbitrarily-nested type constructors but also provides some features not found in Essence, such as the ability to define predicates. The Zinc language is still in development and a full implementation does not yet exist.

To fully realise 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 [Michel and Van Hentenryck, 2000]. It is worth noting that these are all

$$\begin{array}{|l}
\textit{SONET\_Optimisation\_Part} \\
\textit{solution} : \textit{SONET} \\
\textit{objective} : \textit{SONET} \longrightarrow \mathbb{N} \\
\hline
\forall\, s : \textit{SONET} \bullet \textit{objective}(s) = \#(s.\textit{rings-nodes}) \\
\textit{objective}(\textit{solution}) = \min(\textit{objective}(\!|\textit{SONET}|\!)) \\
\hline
\end{array}$$

Figure 2: Part of a Z specification of the SONET problem.

declarative languages. In contrast, programming languages that are augmented with constraint facilities, such as Solver and Eclipse, achieve quantification through iteration, which makes it impossible to support quantification over variables.

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.[5]

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 [2004], 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 [RGU Web, 2006]. 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 the CSP. For example, unlike ESSENCE, nothing prevents a decision variable from having no domain or an infinite domain and nothing prevents using 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 [RGU Web, 2006], of the SONET problem in the two languages. Figure 2 shows part of that Z specification, which is equivalent to the ESSENCE statement "`minimising` $|\textit{rings-nodes}|$".

The Alloy [Jackson, 2006] language 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 terms of relations and atoms and maps these specifications to efficient SAT models. Without a more expressive type system, however, it is not obvious how Alloy could be mapped to constraint languages, and in particular make good use of global constraints, which are vital to efficient constraint models.

8 Conclusion

ESSENCE is a formal language that is natural in that it accessible to someone with an understanding of discrete mathematics but not constraint programming. ESSENCE allows combinatorial problems to be specified at a high level of abstrac-

[5]Frisch and Miguel [2006] argue that the sets of unnamed objects provided by ESRA are a different facility that does not yield all the benefits of unnamed types.

tion. The result is that problems can be specified without (or almost without) modelling them. The central, unique feature of ESSENCE is that it supports complex, arbitrarily-nested types. Consequently, a problem that requires finding a complex combinatorial object can be directly specified by using a decision variable whose type is precisely that combinatorial object.

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. among other developments, we expect that this will lead to the incorporation of additional operators for constructing expressions and additional type constructors such as lists, trees and graphs.

Acknowledgements

Ian Miguel is supported by a UK Royal Academy of Engineeering/EPSRC Research Fellowship. We thank Andy Grayland for his useful comments on the syntax of ESSENCE.

References

[Brooke *et al.*, 1988] Anthony Brooke, David Kendrick, and Alexander Meeraus. *GAMS: A Users' Guide*. The Scientific Press, Danvers, Massachusetts, 1988.

[Cadoli *et al.*, 2000] Marco Cadoli, Giovambattista Ianni, Luigi Palopoli, Andrea Schaerf, and Dominico Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, 26:165–195, 2000.

[de la Banda *et al.*, 2006] Maria Garcia de la Banda, Kim Marriot, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Principles and Practice of Constraint Programming — CP 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 700–705. Springer, 2006.

[Flener *et al.*, 2004] Pierre Flener, Justin Pearson, and Magnus 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.

[Frisch and Miguel, 2006] Alan M. Frisch and Ian Miguel. The concept and provenance of unnamed, indistinguishable types. Available at www.cs.york.ac.uk/aig/constraints/AutoModel/, September 2006.

[Frisch *et al.*, 2005a] Alan M. Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian 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.

[Frisch *et al.*, 2005b] Alan M. Frisch, Brahim Hnich, Ian Miguel, Barbara M. Smith, and Toby 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.

[Frisch *et al.*, 2005c] Alan M. Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In *Proc. of the Nineteenth Int. Joint Conf. on Artificial Intelligence*, pages 109–116, 2005.

[Fuchs and Schwitter, 1996] Norbert Fuchs and Rolf Schwitter. Attempto controlled English (ACE). In *Proc. of 1st International Workshop on Controlled Language*, 1996.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.

[Gent *et al.*, 2006] Ian Gent, Christopher Jefferson, and Ian Miguel. Minion: Lean, fast constraint solving. In *Proceedings of the 17th European Conference on Artifical Intelligence*, 2006.

[Gervet, 1994] Carmen 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.

[Hnich, 2003] Brahim Hnich. *Function Variables for Constraint Programming*. PhD thesis, Computer Science Division, Dept. of Information Science, Uppsala University, 2003.

[Jackson, 2006] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[Lauriere, 1978] Jean-Louis Lauriere. ALICE: A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[Marriott *et al.*, 2006] Kim Marriott, Reza Rafeh, Mark Wallace, Maria Garcia de la Banda, and Nicholas Nethercote. Zinc 0.1: Language and libraries. Technical report, Monash University, 2006.

[Michel and Van Hentenryck, 2000] Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5(1/2):43–84, 2000.

[Renker and Ahriz, 2004] Gerrit Renker and Hatem 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.

[RGU Web, 2006] www.comp.rgu.ac.uk/staff/ha/ZCSP/, 2006.

[Simons, 1987] R.V. Simons. Mathematical programming modeling using MGG. *IMA Journal of Mathematics in Management*, 1:267–276, 1987.

[Spivey, 1989] J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, 1989.

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

[Wallace *et al.*, 1997] M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.

[York Web, 2006] www.cs.york.ac.uk/aig/constraints/AutoModel, 2006.