Why Essence?

Frequently Asked Questions about a New Language for Specifying Combinatorial Problems

Alan M. Frisch[1], Matthew Grum[1], Christopher Jefferson[2], Bernadette Martínez Hernández[1], and Ian Miguel[3]

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

Abstract. Essence is a new language for specifying combinatorial (decision or optimisation) problems at a high level of abstraction. Since its introduction, a number of misconceptions about the language and its purpose have arisen in the constraint community. This FAQ addresses some of the frequently arising questions and tries clarify some common misconceptions.
This document is not intended to serve as an introduction to the Essence language; that is handled elsewhere [6]). Rather, this document is intended to complement such an introduction. Nonetheless, the answer to the first question provides a very brief introduction to the language.

What is Essence?

Essence is a new language for specifying combinatorial (decision or optimisation) problems at a high level of abstraction. Essence is the result of our attempt to design a formal language that enables problem specifications that are similar to rigorous specifications that use a mixture of natural language and discrete mathematics, such as those catalogued by Garey and Johnson [9].

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 [16] or NP-Spec [3]. This has resulted in a language that, to a first approximation, is a constraint language, such as opl [17], $\mathcal{F}$ [12] or esra [4], 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 also provides 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.

Let us give some examples of Essence in order to demonstrate it's naturalness and to briefly introduce the language to a reader who is unfamiliar with it.

First, we ask 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 [9, 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 claim that Essence specifications can be similar to the rigorous specifications that people naturally use. Hence, someone able to understand these rigorous problem specifications that employ discrete mathematics can, with little training, also understand Essence specifications.

Natural problem specifications, such as the one above, identify what is given (the parameters to the problem), what combinatorial object(s) must be found (the decision variable) and what constraints the object(s) 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 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.

Now consider the specification of the Golomb Ruler Problem (GRP, problem 6 at www.csplib.org) in Fig. 1. 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

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 in [7])

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).

1. given w, g, s : int (1...)
2. letting $golfers$ be new type of size $g * s$
3. find $sched$: mset (size w) of rpartition (size s) of $golfers$
4. such that $\forall_{week1, week2 \in sched} . week1 \neq week2 \implies$
$\forall_{group1 \in week1, group2 \in week2} . |group1 \cap group2| < 2$

Alternative constraint:

$4'$. such that $\forall_{golfer1, golfer2 : golfers} . golfer1 \neq golfer2 \implies$
$(\sum_{week \in sched} .\texttt{together}(golfer1,\ golfer2,\ week)) < 2$

Fig. 1. ESSENCE specifications of four problems.

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, which is a type containing a specified number of unnamed elements. 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 constraint at line 4 quantifies over the weeks, ensuring that the size of the intersection between every pair of elements of the corresponding partitions is at most one (otherwise the same two golfers are in a group together more than once). The constraint at line $4'$ quantifies over the pairs of golfers, ensuring that they are partitioned together (via the global constraint `together`) over the weeks of the schedule at most once. Note that here we make use of a facility common to constraint languages: treating Booleans as 0/1 for the purpose of counting.

What motivated you to design Essence?

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 always 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, problem 10 in CSPlib [11]), 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 [4] 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 different ways that this can be done [8].

To automatically generate models for a problem, one must start with a formal specification of the problem and this specification must be sufficiently abstract so 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*

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

specification language and distinguish it from *modelling languages*, whose purpose is enable the specification of models. Thus, designing a problem specification language is a *prerequisite* to studying automated modelling and our primary motivation for designing Essence has been 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 repository 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.

What were the objectives in designing Essence?

Emphasizing points made in answering the previous question, two main objectives influenced the design of Essence: naturalness and abstractness.

Naturalness: The language should be similar to the way in which people naturally think of problems and the style in which they specify them informally. If this is achieved then we believe that the language should be accessible to those with no background in constraint programming, though we believe that some background in discrete mathematics is needed.

Abstractness: The language should provide sufficient abstraction so that modelling decisions do not need to be made in specifying a problem. In particular, one should not have to make arbitrary distinctions in writing a specification.

These two objectives are not unrelated. Providing an appropriate level of abstractness is necessary to achieve naturalness.

What evidence is there that you have met the design objectives?

To evaluate Essence we specified a large suite of problems in the language and here reflect on the process and results. A suite of 60 problems, both theoretical and practical, was selected, 25 drawn from www.csplib.org, and 35 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 [2] 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 large examples being just as easy to read.

Though Essence has been shown to be useful in specifying a moderately large set of problems, how do we know that you haven't selected the problems based on their ease of specification?

The vast majority of problems in CSPLib have been specified in Essence. CSPLib was developed prior to, and independently of the development of Essence.

However, what is true, is that Essence was designed to facilitate the specification of problems of the type that people of have tended to add to CSPLib.

There already exist many specification languages, most notably Z. Do we really need another one?

Another approach to problem specification is to employ a more-powerful, more-general specification language such as B or Z. This approach has been explored thoroughly by Renker and Ahriz [15], 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 [1]. 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 constraint satisfaction problem. 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

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

Fig. 2. Part of a Z specification of the SONET problem.

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 [1], of the SONET problem in the two languages. Figure 2 shows part of that Z specification which is equivalent to the Essence statement "`minimising` $|\textit{rings-nodes}|$".

Essence appears to provide some redundant type constructors. Why?

Essence provides a partition type constructor even though a partition could be obtained by imposing a constraint on a set of sets. And Essence provides a function type constructor even though a function could be obtained by imposing a constraint on a relation (and furthermore, relations can be obtained as sets of tuples). Hence, the set of type constructors provided by Essence is not minimal.

Our primary reason for having such redundant type constructors is that this is useful for our rule-based method for automatic model generation. There are special techniques that can be used for modelling partitions that cannot be used for modelling sets of sets in general. As our modelling rules are driven by the type system, it is useful to reflect such distinctions in the type system.

Furthermore, the distinction between, say, a function and a relation is useful in specifying the syntax of the language. For example, function application can be used for functions but cannot be used for relations in general.

Essence appears to be missing type constructors for some important combinatorial objects. It also is missing some of my favourite operators. Why?

Essence is a language under development and so far only an early version, Version 1.1.0 has been designed. We included enough type constructors to demonstrate the utility of the language for specifying a wide range of problems. The type constructors provided by Essence 1.1.0 are set, multiset, partition, regular partition, tuple, relation, function and matrix. There is no doubt that additional type constructors would be useful, notable examples being lists, graphs and trees. Similar comments apply to the set of operators that has been included in the language.

Our methodology for developing the language is to use it to specify a wide range of problems and identify language enhancements and extensions based on this.

Why does Essence provide a `maxsize` annotation but no `minsize` annotation?

In specifying a domain, 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. For example, `set` (`maxsize` n) `of int` is a legal domain. However, there is no comparable minsize annotation.

The reason for this is that the `maxsize` annotation is often necessary to make a domain finite. For example, omitting (`maxsize` n) from the above domain results in an infinite domain. In contrast, `minsize` could not be used to make a domain finite and is therefore unnecessary. We may, however, ultimately decide that `minsize` is a useful construct and add it to the language

Why is it so important to avoid introducing symmetry into problem specifications?

Symmetry can enter a constraint model from two sources: it can be inherent in the problem and it can be introduced by the modelling process. Modelling languages often force a specification to introduce unnecessary objects or to unnecessarily distinguish between objects, and this typically introduces symmetry into the model.

To see an example of this point, observe that many problems involve some set of elements, yet do not constrain or otherwise mention the particular elements. For example, the Social Golfers Problem does not name the golfers. Hence the golfers are indistinguishable. Yet all constraint languages, other than Essence and ESRA (which adopted unnamed types from Essence), require the golfers to be named. Naming, and hence distinguishing, the otherwise indistinguishable golfers introduces symmetry into the model; namely, in any solution interchanging the golfers' names results in a solution. As the golfers' names are typically values, this is typically a value symmetry.

A language that has sufficient facilities for abstraction would not force the introduction of unnecessary objects or distinctions. Providing such abstraction facilities has been a guiding principle in the design of Essence. One such facility provided by Essence is a type whose elements are unnamed and, hence, indistinguishable. This can be used to model the golfers in the SGP without introducing symmetry into the specification.

As another illustration of this point consider a problem that involves finding an unordered pair of integers. A straightforward model would use a pair of integer decision variables. Such a model would have variable symmetry: in any solution interchanging the the two variables results in a solution. Of course, in a modelling language that supports sets of integers, this symmetry could be eliminated by using a single decision variable whose domain values are sets of integers of cardinality two. But the general problem still remains. The SONET problem, for example, requires placing communication nodes on a set of n communication rings. Thus, the problem requires finding a set (of cardinality n) of sets. In ESSENCE this can be modelled by a decision variable whose domain contains sets of sets. But ESSENCE is the only constraint language that supports sets of sets, so in any other language the problem would typically be modelled by n set variables.[5] And once again, the model has variable symmetry as the n variables can be permuted.

We have used the elimination of unnecessary symmetries as a way of evaluating ESSENCE. So far, every problem we have considered has an ESSENCE specification that contains no symmetries other than those that can be considered to be inherent in the problem. Every other constraint language fails this test.

The elimination of unnecessary symmetries from problem specifications has important consequences for the automation of modelling. As a symmetry that is present in a model is either present in the problem specification or was introduced by the modelling process, removing a symmetry from the problem specification means that it has moved into the modelling process – and this is desirable. We believe that modelling is a systematic process and that the symmetries it introduces are also systematic. Therefore, an automated modelling system ought to be able to automatically identify the symmetries it has introduced into the models it generates. And, indeed, our automated modelling prototype, CONJURE, does just that [5].

Symmetry that remains in a problem specification can be difficult to identify, either by a human or by an automated process. Though significant progress has recently been made on the automatic identification of symmetry, we believe that it is always easier to identify the symmetry when it is introduced.

What is the current status of ESSENCE?

At this point we have a full definition of Version 1.1.0 of the language. We have presented a formal syntax [2] and a formal semantics [6].

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. The parser also performs all necessary category checking. Every term of the language belongs to one of three categories: constants, whose values are determined by the specification; parameters, whose values can only be determined once the instance data is known; and variables, whose values are determined by solving a problem instance. The category checker ensures that a variable term is not used where a constant or parameter term is required; for example, the size of an array cannot be specified by variable terms.

[5] For the purpose of this illustration we are ignoring the possibility of considering the SONET problem as one of finding a relation between the rings and the nodes¿

A second implementation of this same parser in Java is nearing completion.

We have identified a subset of ESSENCE 1.1.0, called ESSENCE′ 1.b.a, 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 translator for mapping ESSENCE′ to Eclipse [18] and are currently developing another one to map ESSENCE′ to Minion [10].

What do ESSENCE and ESSENCE′ have to do with automated modelling?

To automate constraint modelling we need a formal definition of what the modelling process is.

ESSENCE is a formal language in which problems can be specified without making modelling decisions. ESSENCE′ is a formal language for specifying models in which all modelling decisions have been made. Taken together, these two languages give us a *formal* definition of constraint modelling:

> *Constraint modelling is the process of translating an* ESSENCE *specification into an equivalent*[6] ESSENCE′ *specification.*

There are usually many, and sometimes a huge number, of ESSENCE′ models that are equivalent to a given ESSENCE specification. Some of the ESSENCE′ models will be better than others at solving the problem. The goal of *effective* modelling is to translate an ESSENCE specification into one, or some, of the better models.

What progress have you made towards the automation of modelling?

The automation of modelling is an extremely ambitious goal, one which we believe can form the basis of a life-long project. We have so far only tackled a piece of the problem, but we believe that it is an important piece and that our progress on this is encouraging.

We have begun by distinguishing two aspects of the study of modelling: the *generation* of correct models and the *selection* of the better models from among the correct ones. This closely corresponds to the distinction in linguistics between competence and performance, a distinction which has served the field well. Most

[6] This paper will not define what it means for two specifications to be equivalent, but note that the formal semantics of the languages is necessary to formulate such a definition.

of our effort has been focussed on model generation, so we focus on that here. However, before proceeding with that, we note that some progress has been made towards developing a theory of when and why some models are better than others [13].

Our main goal is to formulate, and implement, a set of recursive rules that can translate — we say refine — an Essence specification into a set of reasonable Essence′ models. The set of models generated by the rules should include all the "better" models, though it may also include other correct models. So far we have developed a set of rules — called refinement rules — that can refine a fragment of Essence called mini-Essence. This fragment contains only two type constructors: sets and matrices. Though mini-Essence is quite small, it permits arbitrarily-deep nesting of types — sets, sets of sets, sets of sets of sets, ... — which is the language feature that presents the most formidable challenge for refinement. Our refinement rules generate model *kernels* — that is models that are correct but do not contain many of the enhancements that characterize the most effective models, such as symmetry-breaking constraints and implied constraints.

Our refinement rules for mini-Essence have been implemented in a Haskell program called mini-Conjure. As with most speculative software projects, we embarked on this knowing that our first version would be ad hoc and would eventually be thrown away when we learned how to construct the system in a cleaner, more-structured way. We have reached that point and are just now completing the reimplementation of mini-Conjure.

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.* [8] 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 mini-Conjure, we believe we have all the technology needed to refine all of Essence to model kernels.

Of course, our ultimate goal is to develop a refinement system that can refine any Essence specification into a single, complete, effective Essence′ model.

What are your plans for future work?

We plan to specify a much wider range of problems in Essence and to use the insight gained from this to further develop the language.

Most of our effort will be focused on automated modelling. We will fully develop Conjure so that it can generate model kernels for the full Essence language. We will also work on the automated generation of complete models, concentrating on the automated generation of symmetry-breaking constraints and continuing our work on the automated generation of channelling constraints [14].

We would also like to attempt to use the techniques developed in Conjure to develop a system that generates models for other types of solvers, such as integer linear programs, SAT, or pseudo-Boolean formulations.

References

1. www.comp.rgu.ac.uk/staff/ha/ZCSP/.
2. www.cs.york.ac.uk/aig/constraints/AutoModel/.
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. 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.
5. A. M. Frisch. Symmetry and the generation of constraint models. Invited talk at the *Workshop on Automated Reasoning*, July 2005. Slides available at www.cs.york.ac.uk/aig/constraints/AutoModel/.
6. A. M. Frisch, M. Grum, C. Jefferson, B. M. 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.
7. 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.
8. A. M. Frisch, C. Jefferson, B. M. Hernández, and I. Miguel. The rules of constraint modelling. In *Proc. of the Nineteenth Int. Joint Conf. on Artificial Intelligence*, pages 109 – 116, 2005.
9. M. Garey and D. Johnson. *Computers and Intractability.* W. H. Freeman, 1979.
10. I. Gent, C. Jefferson, and I. Miguel. Minion: Lean, fast constraint solving. In *Proceedings of the 17th European Conference on Artifical Intelligence*, 2005.
11. I. P. Gent and T. Walsh. CSPLib: A problem library for constraints. `www.csplib.org`, 2005.
12. B. Hnich. *Function Variables for Constraint Programming.* PhD thesis, Computer Science Division, Dept. of Information Science, Uppsala University, 2003.
13. C. Jefferson and A. M. Frisch. Representations of sets and multisets in constraint programming. In *Proc. of the 4th Int. Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 102–116, 2005.
14. B. Martínez Hernández and A. M. Frisch. The systematic generation of channelling constraints. In *Proc. of the 4th Int. Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 89–101, 2005.
15. 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.
16. J. M. Spivey. An introduction to Z and formal specifications. *Softw. Eng. J.*, 4(1):40–50, 1989.
17. P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint programming in OPL. In G. Nadathur, editor, *Proceedings of the Principles and Practice of Declarative Programming, International Conference PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 98–116, Paris, France, September 29 - October 1 1999. Springer.
18. M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.