Specifying Constraint Problems with Z

Hatem Ahriz and Ines Arana

School of Computing, The Robert Gordon University, Aberdeen, UK

Abstract. In this paper, we present a novel use of the formal specification language Z for the construction of constraint problem models. These models can then be refined into constraint programs. Z's expressiveness and suitability is proven by contrasting Z with three existing abstract constraint languages and by, successfully and efficiently, specifying numerous constraint and optimisation benchmark problems in Z. These specifications have been published in an online library, which also contains three toolkits which support the specification of a variety of problem types.

1 Introduction

The formal specification language and publicly available standard Z [1] can be usefully used to integrate constraint modelling in the standard software design cycle [2]. In this paper, we further show that Z is on a par with, if not more powerful than, current abstract constraint languages such as OPL, $\mathcal{F}$, ESRA and ESSENCE (discussed later). The main difference that distinguishes our line of research from these languages is the fact that Z is a true *specification* language, whereas the mentioned languages are de facto *implementation* languages whose abstraction level has been increased to varying degrees. The benefits of seperating specification from implementation in modelling constraint problems, discussed in detail in [2], can be summarised as follows: (a) a formal specification can serve as a means of communication within a modelling community, (b) it can also interact with computer tools to help prove and verify that a constraint problem is, syntactically and semantically, covered by a model, (c) and finally, it allows higher-level abstractions of model formulations.

The remainder of this paper is structured as follows. After a summary of relevant features of Z in section 2, we briefly show in section 3 how to use Z to model constraint problems. In section 4, we contrast Z with three existing abstract constraint languages. We then discuss how Z specifications can be refined into constraint programs in section 5.

2 A Brief Summary of Z

Z is a typed formal specification language based on first-order logic and Zermelo-Fraenkel set theory. Z has been standardised as ISO/IEC standard 13568:2002. In this section we shall only present the key features of Z, and refer to [1] for further details. The main elements of a Z specification are given sets, axiomatic definitions and schemas. *Given sets* are uninterpreted name references to sets. For example,

$[Warehouses, Suppliers]$

introduces the sets *Warehouses* and *Suppliers* with a scope global to the specification. An *axiomatic definition* also has a global scope and consists of a declaration part and an optional predicate part separated by a horizontal line. The main use of axiomatic definitions is for introducing constants and constant mappings. For example, to define a constant mapping *square* we can write:

$$square : \mathbb{N} \longrightarrow \mathbb{N}$$
$$\forall n : \mathbb{N} \bullet square(n) = n * n$$

Several type constructors, e.g. tuples, Cartesian product and (finite) power-sets, are provided by default, as well as common mathematical data types such as relations, functions, sequences and bags. A *schema* is used in order to design composite data types. Like axiomatic definitions, schemas consist of a declaration and a predicate part, but declared constants and variables are locally-scoped. A schema defines a type which can be referenced via the name of the schema. The schema below can be referred to as *SQPAIR*:

SQPAIR
$$x, y : \mathbb{N}$$
$$y = square(x)$$

Schemas can be combined into new ones using the *operators* of the schema calculus such as logical connectives and quantifiers.

A *type* in Z is either a given set or a composite type. A *composite type* is one of: (a) the *powerset* of a type, (b) the *Cartesian product* of several types, (c) the *schema product type* of several types.

The schema type is expressed as $\langle\!| x_1 : T_1; \ \ldots; \ x_n : T_n |\!\rangle$ where x_i is a variable identifier and T_i its type.

3 Z-Based Constraint Modelling

Constraint satisfaction problems (CSPs) are usually defined as a triple (X,D,C) of variables X, domains D and constraints C formulated over X. In the majority of constraint (logic) programming languages, the constraints in C can be expressed as quantified formulae of first-order logic. This allows a representation of constraint satisfaction problems in Z by single schemata where the elements of X and D are contained in the declaration part and the constraints in C in the predicate part:

CSP
$$x_1 : D_1; \ \cdots; x_n : D_n$$
$$\ldots \ constraints \ \ldots$$

The specification of constrained optimisation problems (CSOPs) in Z uses schemas as types of variables. Let the constraints of a CSOP $\mathcal{P}$ be expressed by a schema *CSP* as before. Then a generic way of specifying $\mathcal{P}$ in Z is via the following schema.

$$
\begin{array}{|l}
\textit{CSOP} \\
\hline
\textit{objective} : \textit{CSP} \longrightarrow \mathbb{Z} \\
\textit{solution} : \textit{CSP} \\
\hline
\forall\, c : \textit{CSP} \bullet \textit{objective}(c) = \ldots \\
\textit{objective}(\textit{solution}) = \min(\textit{objective}(\!|\textit{CSP}|\!)) \\
\hline
\end{array}
$$

In the above we have supposed minimisation. For maximisation replace $\max$ by $\min$.

3.1 Example

To illustrate the specification of CSPs and CSOPs, we use the bus driver scheduling problem (`prob022` of the CSPLib benchmark library [3]).

We are given a set of tasks and a set of shifts, each covering a subset of the tasks and each with an associated, uniform cost. The shifts need to be partitioned such that each task is covered exactly once and the objective is to minimise the total number of shifts required to cover the tasks. The sets of interest are *tasks* and *shifts*,

$$
\begin{array}{|l}
\textit{tasks}, \textit{shifts} : \mathbb{F}\, \mathbb{N} \\
\end{array}
$$

defined here as finite sets of natural numbers. A driver schedule is defined as follows:

$$
\begin{array}{|l}
\textit{Driver\_Schedule} \\
\hline
\textit{coverage} : \textit{shifts} \longrightarrow \mathbb{F}\, \textit{tasks} \\
\textit{allocate} : \mathrm{iseq}\, \textit{shifts} \\
\hline
\textit{coverage} \circ \textit{allocate}\ \mathsf{partition}\ \textit{tasks} \\
\hline
\end{array}
$$

The function *coverage* is part of the instance data and denotes the possible subsets of *tasks*. The only decision variable is *allocate*, an injective sequence of *shifts* (shifts can appear at most once).

The composition of *allocate* with *coverage* yields a sequence of subsets of *tasks*. The built-in partition operator of Z [1, p. 122] asserts that this sequence of subsets is a partition of the set *tasks*. We continue with the *optimisation part* of the problem.

$$
\begin{array}{|l}
\textit{Optimisation\_Part} \\
\hline
\textit{objective} : \textit{Driver\_Schedule} \longrightarrow \mathbb{N} \\
\textit{solution} : \textit{Driver\_Schedule} \\
\hline
\forall\, ds : \textit{Driver\_Schedule} \bullet \textit{objective}(ds) = \#(ds.\textit{allocate}) \\
\textit{objective}(\textit{solution}) = \min(\textit{objective}(\!|\textit{Driver\_Schedule}|\!)) \\
\hline
\end{array}
$$

The *objective* function maps each element of the solution set *Driver_Schedule* into the number of shifts represented as the cardinality of the *allocate* variable. The last expression states that the element *solution* of the solution set must have the minimal value for the *objective* function.

3.2 The Online Library

We omit further examples since part of this research work is a comprehensive online library [4] containing 67 fully specified problem instances, including 33 benchmark problems from CSPLib [3], as well as three comprehensive constraint toolkits. Each problem specification introduces the problem, often discusses a concrete implementation and data examples in addition to literature references, and presents one or more fully specified Z models. Every Z specification has been type-checked (using the fuzz type checker [1] and the Z-Eves prover [2]). The LaTeX source code is made available for further processing.

The toolkits provide common abstractions for constraint modelling, concrete support for the scheduling domain, and a toolkit for global constraints.

4 Z vs. Other Abstract Languages

In this section we study three abstract constraint languages and contrast their available data types, vocabulary and constraint expressions with corresponding facilities in Z. This serves to demonstrate that Z is on a par with, if not more powerful than, current abstract constraint languages.

4.1 The $\mathcal{F}$ language

The salient feature of the $\mathcal{F}$ language [5] is the use of *constrained function variables*. Several of the other language features were inspired by OPL (Optimization Programming Language) [6]. Basic *data types* of $\mathcal{F}$ are integer and set of integers. A Cartesian product operator ($\times$) is provided for the construction of compound types.

Expressions and formulae in $\mathcal{F}$ are built from constants, input and decision variables. The basic operators in $\mathcal{F}$ have obvious Z counterparts.

As in OPL, *decision variables* are identified using the keyword `var`. Table 1 summarises the elements that are peculiar to the $\mathcal{F}$ language and relates each of these to Z's counterpart. In these declarations, F is a function variable and W can be of type integer or an integer set type.

The parenthesised expressions in the second-last and third-last row of table 1 refer to *partial* surjections and injections respectively. Table 2 shows the *sequence* constructs in $\mathcal{F}$ and their representation in Z. In $\mathcal{F}$, there is a distinction between sequences of fixed and bounded length. In Z, the sequence length can be enforced via an additional constraint.

[1] http://spivey.oriel.ox.ac.uk/mike/fuzz/ (accessed 12 Apr 06)
[2] http://www.ora.on.ca/z-eves/ (accessed 12 Apr 06)

Table 1. Function variables in $\mathcal{F}$ and their counterpart in Z

Concept	$\mathcal{F}$ expression	Z expression
F is a total function	$F : V \longrightarrow W$	$F : V \longrightarrow W$
F is a partial function	$F : V \longmapsto W$	$F : V \pfun W$
Function application	$F(i)$	$F(i)$
Set of elements that have image j under F	$F^{-1}(j)$	$F^{\sim}(\!\mid\{j\}\mid\!)$
Membership	$\langle i, j\rangle \in F$	$(i, j) \in F$
$\{i \in V \mid (\exists j \in W) \bullet (i, j) \in F\}$	$\mathit{domain}(F)$	$\mathrm{dom}\, F$
$\{j \in W \mid (\exists i \in V) \bullet (i, j) \in F\}$	$\mathit{range}(F)$	$\mathrm{ran}\, F$
F is surjective	$\mathit{surjective}(F)$	$F : V \fsurj W$ $(F : V \pfsurj W)$
F is injective	$\mathit{injective}(F)$	$F : V \finj W$ $(F : V \pfinj W)$
F is bijective	$\mathit{bijective}(F)$	$F : V \fbij W$

Table 2. Sequences in $\mathcal{F}$ and their representation in Z

Concept	$\mathcal{F}$ expression	Z declaration	Z constraint
fixed-length sequence	$S : \mathit{seq}(V, k)$	$S : \mathrm{seq}\, V$	$\#S = k$
bounded-length sequence	$S : \mathit{bseq}(V, k)$	$S : \mathrm{seq}\, V$	$\#S \leq k$
length of a sequence	$\mathit{length}(S)$	$\#S$	
i-th element of S	$S(i)$	$S(i)$	
range	$\mathit{range}(S)$	$\mathrm{ran}(S)$	
permutation P over set V	$P : \mathit{perm}(V)$	$P : \mathrm{iseq}\, V$	$\mathrm{ran}(P) = V$

We omit specific examples of the $\mathcal{F}$ language since detailed comparisons of Z and $\mathcal{F}$ models of the following problems can be found in the online library [4]: graph colouring problem, `prob006`, `prob007`, `prob015`, `prob022`, `prob024`, `prob030`, and `prob034` of CSPLib.

4.2 The ESRA language

ESRA (*Executable Symbolism for Relational Algebra*) [7] is a constraint language which builds upon previous work on extensions for OPL and $\mathcal{F}$. ESRA is called a *relational language* as it allows relations and functions to appear as decision variables. The main characteristics of ESRA are that it provides: (a) a data type for (functional) relations, (b) finite-domain constraints only, (c) no sequences, nested sets or multisets, (d) no negation, recursion or unbounded quantification. A program (called a *model*) is a sequence of domain, constant, and decision-variable declarations, followed by a constraint section; all without separators. Figure 1 shows a model for `prob028` [4].

Domain declarations are marked by the keyword `dom` and specify named subsets of the predefined numerical domains $\mathbb{N}$ and $\mathbb{Z}$ (called `nat` and `int`, respectively). A *primitive domain* can be an extensionally or an intensionally specified set, or a set of names.

`dom` *Varieties*, *Blocks*
`cst` $r, k, \lambda \in \mathbb{N}$
`var` *BIBD* : *Varieties* $^{r}\times^{k}$ *Blocks*
`solve`
$\forall(v_1 < v_2 \in \mathit{Varieties})\mathit{count}(\lambda)(i \in \mathit{Blocks} \mid \mathit{BIBD}(v_1, i) \wedge \mathit{BIBD}(v_2, i))$

Fig. 1. Balanced Incomplete Block Design (BIBD) in ESRA

Intensionally specified sets can be introduced using set comprehension or as integer ranges using the $a \ldots b$ notation in analogy to Z. *Compound domains* use one of the constructors from table 3.

Table 3. Relations in ESRA

Concept	ESRA expression
General relation	A $^{M_1}\times^{M_2}$ B
Partial function	A $\not\longrightarrow^{M_2}$ B
Total function	A $\longrightarrow^{M_2}$ B
Subset variable	$A\,M_1$

The *product constructor* A $^{M_1}\times^{M_2}$ B creates a *relation* over the sets A, B. Each of the identifiers M_1, M_2 denotes either a single integer (cf. table 3) or a set of integers. The semantics of the product constructor are as follows. For every element $a \in A$, the number of elements $b \in B$ that are related to a must be in (or equivalent to) M_1 and conversely, for every $b \in B$, the number of elements $a \in A$ that are related to b must be in (or equivalent to) M_2. When omitted, M_1, M_2 default to $\mathbb{N}$. The next two constructors in table 3 are specialisations of the product constructor which denote *functional relations*, where M_2 retains its sense. A further special case is the *subset variable constructor* which denotes the set of all subsets of a given set A whose cardinalities appear as elements of M_1.

Constant declarations can be used for instance data. Constants are declared after the `cst` keyword and are related to the corresponding domains using either '$\in$' for scalars or ':' (with the meaning of $\subseteq$) for sets and relations.

Variable declarations are, as in OPL, identified by the keyword `var` and follow the same conventions as the declarations of constants. In addition, a projection clause can be adjoined to the declaration of a relation variable.

Constraint declarations appear as quantified formulae, using the operators and connectives shown in table 4.

Quantified statements are similar to Z's, where S is a set expression and P an optional predicate part. The expression $\mathrm{card}(S)$ corresponds to $\#S$ in Z. For *cardinality constraints*, ESRA provides the syntactic sugar $\mathit{count}(M_1)(x \in \mathit{Set} \mid \mathit{Condition})$ in place of the equivalent expression $\mathit{card}(\{x \in \mathit{Set} \mid \mathit{Condition}\}) \in M_1$. The *aggregate operator* Σ iterates over a set expression $S \mid P$, summing the values of the expression f for the elements of this set.

Table 4. Basic expressions in ESRA

Arithmetic	$+,\ -,\ *,\ /,\ \%$, abs
Set-based	$\mathrm{card}(S),\ \ ^{a}\!\times^{b},\ \ \longrightarrow, \not\longrightarrow$
Relational	$<,\ \leq,\ =,\ \neq,\ \geq,\ >$
Logic	*true*, *false*, $\wedge,\ \vee,\ \Rightarrow,\ \Leftarrow,\ \Leftrightarrow$
Quantified	$\forall S \mid P,\ \ \exists S \mid P$
Aggregate	$\Sigma_{S\mid P}\, f$, *count*

Expressing ESRA within Z All of the basic ESRA constructs (table 3 and 4) have immediate counterparts in Z, or can be defined within Z and are provided in one of our toolkits. For example, we define the *absolute value function* as follows:

$$\mathrm{abs} \ == \ \ \lambda z : \mathbb{Z} \bullet \mathbf{if}\ z \geq 0\ \mathbf{then}\ z\ \mathbf{else}\ -z$$

The *aggregate operator* $\Sigma_{S|P}\, f$ sums up the images under f of each element in the set specified by the expression $S \mid P$. In considering such an operator, it is important to note that the collective whole of these images forms a *bag* rather than a set (since f is not necessarily injective). We have therefore realised the following modular solution in our toolkit:

1. a '*bag mirroring*' operator $\|\_\|$ which does the same as relational image, but honours repeated values by returning a bag $f\,\|S\|$ which contains $f(s)$ for all $s \in S$
2. a *bag sum operator* $\Sigma : \mathrm{bag}\,\mathbb{Z} \longrightarrow \mathbb{Z}$

The details are omitted here and can be found in the main toolkit of our online library. Next is the *product operator* (table 3) to construct binary relations from two given sets. We first constrain M_1, M_2 to be finite sets of natural numbers, using an axiomatic definition.

$$\mid\ M_1, M_2 : \mathbb{F}\,\mathbb{N}$$

The product constructor can be defined in Z using a *generic schema*, where A, B are passed as schema parameters. The constructor $\_\ ^{-}\!\times^{-}\ \_$ in the declaration section is defined to be a finite function whose return type is a relation $R : A \leftrightarrow B$.

$$\begin{array}{|l}
[A, B] \\
\hline
_\ ^{-}\!\times^{-}\ _ : (\mathbb{P}\,A \times \mathbb{F}\,M_1 \times \mathbb{F}\,M_2 \times \mathbb{P}\,B) \twoheadrightarrow (A \leftrightarrow B) \\
\hline
\forall R : A \leftrightarrow B \ \bullet\ R = A\ ^{M_1}\!\times^{M_2}\ B \Leftrightarrow \\
\qquad \forall a : \mathrm{dom}\,R;\ b : \mathrm{ran}\,R \ \bullet\ \#(\{a\} \lhd R) \in M_1 \ \wedge \\
\qquad \#(R \rhd \{b\}) \in M_2 \\
\hline
\end{array}$$

The *multiplicity constraints* on this relation are declared as cardinality constraints on the respective domain and range restrictions of that relation. Since the definitions of (partial) *functions* do not add any new information, we use short abbreviation definitions in terms of the above schema:

$$A \longrightarrow^{M_2} B == A \ {}^{\{1\}}\!\times^{M_2} B$$
$$A \not\longrightarrow^{M_2} B == A \ {}^{\{0..1\}}\!\times^{M_2} B$$

Last is the definition of the *subset variable constructor* for a given set A.

$$[A]$$
$$\_\ \_ : (\mathbb{P}\, A \times \mathbb{F}\, M_1) \longrightarrow \mathbb{P}\, A$$
$$\forall S : \mathbb{P}\, A \mid S = A\, M_1 \bullet \#S \in M_1$$

Further demonstration of the mapping of ESRA into Z can be found in our online library for `prob010`, `prob013`, and `prob028` [4].

4.3 The ESSENCE language

ESSENCE is an abstract constraint language used to write specifications for automated refinement by the CONJURE system [8]. A number of ESSENCE examples are available online [3].

An ESSENCE specification is structured in a manner similar to that of ESRA, consisting of six different sections (some optional, multiple sections of the same type allowed) introduced by keywords, which are listed in table 5.

Table 5. Sections of an ESSENCE specification

Keyword	Section contents	Optional
`given`	parameters, instance data	•
`where`	restrictions on parameters	•
`letting`	constants	•
`find`	decision variables	
`[mini|maxi]mising`	objective function	•
`such that`	problem constraints	•

Figure 2 shows the Golomb ruler problem `prob006` [4] in ESSENCE.

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

Fig. 2. Golomb ruler problem in ESSENCE

[3] http://www.cs.york.ac.uk/aig/constraints/AutoModel/ (accessed 12 Apr 06)

Elementary data types in ESSENCE have obvious counterparts in Z. Table 6 shows the compound data types of ESSENCE and their representation in Z. These compound data types have several different modes of representation in Z. For instance, our main toolkit supports bounded sets as well as a variety of matrix types, cf. online library.

Table 6. Compound types of ESSENCE and their representation in Z

ESSENCE	Z declaration	Z constraint
`S: set(size` n`) of` T	$S : \mathbb{P}\, T$	$\#S = n$
`S: set(maxsize` n`) of` T	$S : \mathbb{P}\, T$	$\#S \leq n$
`M: mset(size` n`) of` T	$M : \mathrm{bag}\, T$	$\mathit{size}(M) = n$
`M: mset(maxsize` n`) of` T	$M : \mathrm{bag}\, T$	$\mathit{size}(M) \leq n$
`P: partition of` T	$P : \mathrm{seq}\, T$	P partition T
`F :A` $\rightarrow$ `B`	$F : A \pfun B$	
`M: matrix[indexed by` $I_1, \ldots, I_n$`] of` T	$M : I_1 \times \cdots \times I_n \longrightarrow T$	

Problem parameters are introduced by the `given` keyword and declare the types of instance data for the problem. Concrete instance data values are not part of the specification and are, as in ESRA and OPL, to be provided elsewhere. The `where` clause can provide further assertions and restrictions on the input data. The `letting` section is similar to the `cst` section in ESRA where assignment of values to constants uses the keyword `be`. The `find` section declares decision variables and hence serves a similar purpose to the `var` keyword used in OPL, $\mathcal{F}$ and ESRA. Expressions are built from the operators shown in table 7; '$|\_|$' is used for cardinality as well as absolute value and '$\_^{-}$' denotes exponentiation.

Table 7. Basic expressions in ESSENCE

Arithmetic	$+$, $-$, $*$, $/$, $\lvert\_\rvert$, $\_^{-}$
Set-based	$\lvert S\rvert$, $\subset$, $\subseteq$, $\cap$, $\cup$
Relational	$<$, $\leq$, $=$, $\neq$, $\geq$, $>$
Logic	$\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\leftarrow$
Quantified	$\forall$, $\exists$
Aggregate	Σ, *min*, *max*, $\mathrm{AllDifferent}$

Expressing ESSENCE within Z The expression of ESSENCE constructs in Z is straightforward. Representation of compound data types is shown in table 6. Absolute value, exponentiation, integer division, as well as the sum operator Σ and an alldifferent macro are part of our toolkit.

In the ESSENCE repository, a variant `P: partition(size n) of T` is also used. This can be realised in Z with the additional constraint $\forall p : \operatorname{ran} P \bullet \#p = n$.

The specification of the SONET problem in the online library contains a detailed study of an ESSENCE model in contrast with the corresponding Z model.

4.4 Advantages of Z over the compared languages

The comparisons presented in section 4, show that not only does Z allow for very concise and straightforward constructs in the compared languages, but that it also allows for its extension, either on the type side by declaring recursive free types, or on the operator side, using generic axiomatic definitions. Z's inherent extensibility through the use of user-definable toolkits such as those we make available on-line, and illustrated in figure 3, gives it an advantage over the compared languages. Finally, one of the main differences that distinguishes our line of research from these languages is the fact that Z is a true *specification* language, whereas the mentioned languages are de facto *implementation* languages whose abstraction level has been increased to varying degrees.

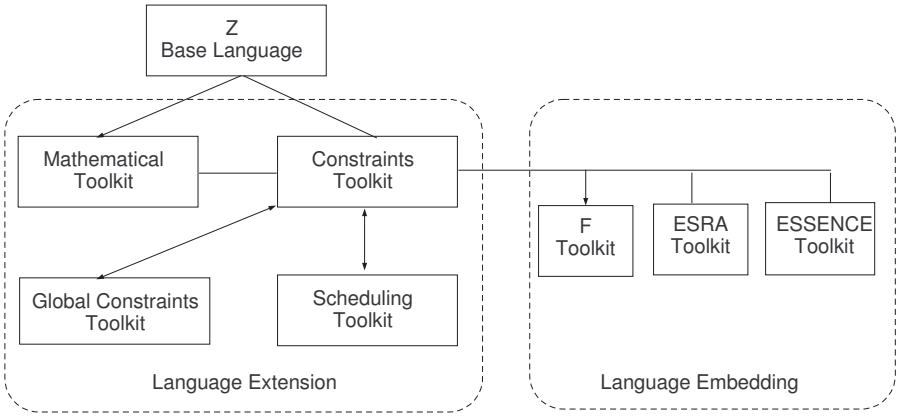

Fig. 3. The use of toolkits for extending and embedding the base language

5 Refining Z Specifications

Refinement in general involves finding suitable ways of *implementing* the specified constraint problem model. For instance, it may not be necessary to refine a specification which involves sequences if the target platform supports constrained array variables. The beauty of choosing Z is that these aspects of refinement can be resolved within Z itself. All that is necessary is to identify the desired data types and then refine the *abstract specification* into a more *concrete specification* supporting the target data types as illustrated in figure 4.

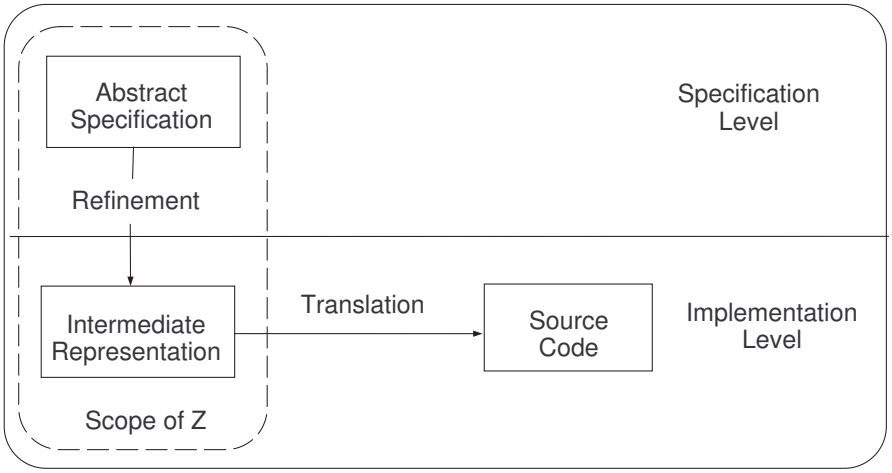

Fig. 4. Specification and Refinement in Z

This entails proof obligations that have to be discharged in order to show that both specifications have the same models. In smaller cases this may be obvious, and in larger cases it can be assisted by proof environments such as Z-Eves. With the same technique we can also generate, in the interest of *redundant modelling*, multiple concrete refinements of the same specification and decide upon the channelling constraints. This aspect shows one of the benefits of using abstract specifications. It is much to the benefit of this work that currently the literature on refinement is proliferating. Hence, in discussing refinement options, we will provide pointers to the associated literature.

1. Refining Sets: If the target language supports set variables (e.g. Ilog Solver, or Eclipse) then refinement of set-based constructs is immediate. Otherwise, there exist two common ways of refining set variables, the characteristic function representation and the explicit representation [9].
2. Refining Relations: The refinement of relation variables is investigated in several places of the literature, e.g. in [10]. If the target language supports n-dimensional arrays of constrained decision variables, table 8 shows the options for refining a relation $R \subseteq S_1 \times \cdots \times S_n$: an n-dimensional array M_n representing the characteristic function χ_R of R, or an $(n-1)$–dimensional array SM_n ranging over subsets of S_n. The particular case of *binary relations* is shown in table 9. The array names were chosen with regard to refining functions, which use the same three options:
   - a 1-dimensional array Fd_1 of set variables indexed by A.
   - a 2-dimensional array Fd_2 representing the characteristic function χ_R.
   - a 1-dimensional array S of set variables indexed by B.
3. Refining Functions: Hnich studied the refinement of various types of functions, and of sequences as special case, in depth [5]. The refinement choices for a function

Table 8. Refinement of an n-ary relation R

Array	Index	Range	Condition
M_n	$S_1 \times \cdots \times S_n$	$\{0,1\}$	$\mathsf{M}_n[s_1]\cdots[s_n] = \chi_R(s_1,\ldots,s_n)$
SM_n	$S_1 \times \cdots \times S_{n-1}$	$\mathbb{P}\, S_n$	$(s_1,\ldots,s_n) \in R \Leftrightarrow s_n \in \mathsf{SM}_n[s_1]\cdots[s_{n-1}]$

Table 9. Refinement of a binary relation $R \subseteq A \times B$

Array	Index	Range	Condition
Fd_1	A	$\mathbb{P}\, B$	$\mathsf{Fd}_1[a] = \{b \in B \mid (a,b) \in R\}$
Fd_2	$A \times B$	$\{0,1\}$	$\mathsf{Fd}_2[a,b] = \chi_R(a,b)$
S	B	$\mathbb{P}\, A$	$\mathsf{S}[b] = \{a \in A \mid (a,b) \in R\}$

$f : A \longrightarrow B$ are in principle the same as for binary relations, with the following differences:

- Fd_1 is a conventional array, ranging over B instead of $\mathbb{P}\, B$.
- The Fd_2 representation requires a constraint $1 = \sum_{b \in B} \mathsf{Fd}_2[a,b]$ for each $a \in A$.
- S is required to be a partition of A (S partition A in Z).

4. Refining Multisets: The type of a bag S in Z is that of a *partial function* from S into $\mathbb{N}_1$. It can be represented by a *total function* $b_S : S \longrightarrow \mathbb{N}$ such that $b_S(x) = 0$ if x does not appear in the bag. Table 10 presents the most common bag constructs in Z [1], where x is any element of S.

Table 10. Notation for common bag operations

Z Syntax	Semantics	Name
$[\![x_1,\ldots,x_n]\!]$	B	bag display
$B \sharp x$	$b_S(x)$	occurrence count
$x \sqsubseteq B$	$b_S(x) > 0$	bag membership
$B_1 \sqsubseteq B$	$(\forall x \in S)\ B_1 \sharp x \leq B \sharp x$	sub-bag relationship

6 Related Work

The use of publicly available software standards to support constraint problem modelling, and of Z in particular, is novel. The only related literature reference that could be found is [11], which uses Z to define the semantics of object-oriented constraint systems in the configuration domain. Apart from pursuing different objectives, the main difference to [11] is that our approach employs Z to specify the problem as such, independent of problem domain (configuration) and programming paradigm (object orientation).

7 Conclusion

In this paper, we have discussed the successful use of Z as an abstract and expressive specification language for CSOPs. We have considered three recent abstract constraint languages and contrasted these with the proposed use of Z. In each case it was possible to easily express the constructs of these languages within Z. This confirms that using Z as a basis for specifying constraint problems offers *adequate and expressive support*. It is interesting to observe that many of the currently favoured abstractions have been used before. As early as 1978, the ALICE system offered support for Cartesian product, set, relation, function, and matrix types, among others [12]. A strong argument in favour of Z in this context is that such abstractions are among the elementary language primitives (a matrix type is provided by our toolkit), and other constructions can be assembled from primitives. A wide variety of benchmark CSOPs have been specified using Z and contrasted with other language specifications. All our Z specifications have been made available online. Finally, we discussed how Z models can be refined into models which are closer to the implementation language. Overall, this paper presents a strong argument for the use of Z to specify constraint problems.

References

1. Spivey, J.: The Z notation: A reference manual. Oriel College, Oxford (1998)
2. Renker, G., Ahriz, H.: Building models through formal specification. Proc. CP-AI-OR04 (2004) 395–401
3. Gent, I., Walsh, T.: CSPLib: a benchmark library for constraints. Technical report APES-09-1999 (1999) http://www.csplib.org/
4. Renker, G.: An online library of Z CSOP models. (2005) http://www.comp.rgu.ac.uk/staff/ha/ZCSP/
5. Hnich, B.: Function variables for constraint programming. PhD thesis (2003) Uppsala University, Sweden
6. Hentenryck, P.: The OPL optimization programming language. MIT (1999)
7. Flener, P., Pearson, J., Argen, M.: Introducing ESRA, a relational language for modelling combinatorial problems. Proc. Reform-03 (2003) 63–77
8. Frisch, A.M., Jefferson, C., Martinez Hernandez, B., Miguel, I.: The rules of constraint modelling. Proc. IJCAI-05 (2005) 109–116
9. Jefferson, C., Frisch, A.M.: On the effectiveness of set and multiset representations in constraint programming. Proc. Reform-04 (2004) 125–141
10. Frisch, A.M., Hnich, B., Miguel, I., Smith, B.M., Walsh, T.: Towards CSP model reformulation at multiple levels of abstraction. Proc. Reform-02 (2002) 42–56
11. Henocque, L.: Modeling object oriented constraint programs in Z. RACSAM (Revista de la Real Academia De Ciencias serie A Mathematicas) **98(1)** (20048) 127–152
12. Lauriere, J.L.: A language and a program for stating and solving combinatorial problems. Artificial Intelligence **10(1)** (1978) 29–127