Unification and multiple views of data in Z

Eerke Boiten, John Derrick, Howard Bowman and Maarten Steen
Computing Laboratory, University of Kent
Canterbury, CT2 7NF, UK.
Phone +44 1227 827553
Email: {eab2,jd1,hb5,mwas}@ukc.ac.uk *

Abstract

This paper discusses the unification of Z specifications, in particular specifications that maintain different representations of what is intended to be the same datatype. Essentially, this amounts to integrating previously published techniques for combining multiple *viewpoints* and for combining multiple *views*. It is shown how the technique proposed in this paper indeed produces unifications, and that it generalises both previous techniques.

1 Why unification?

There is a wide interest in multiple-viewpoint specification in software engineering [FGH+94, Jac94, ZJ93]. This method allows different specifiers to work independently, and to observe systems from various perspectives. Objects and behaviours in the system that are of interest to multiple viewpoints may have overlapping descriptions in each of those viewpoints. Due to the independence of viewpoints, however, the various descriptions need not be identical, and may even turn out to be contradictory, i.e. inconsistent. One definition of consistency is that two specifications are consistent whenever a common implementation of them exists. A constructive way of checking this is by constructing a *unification* of two viewpoints. The unification of two specifications A and B is the most general specification C such that all implementations that satisfy C also satisfy both A and B.

Our particular interest in this subject arises from the Open Distributed Processing Model [ITU95]. In our project 'Cross Viewpoint Consistency in Open Distributed

*This work was partially funded by the U.K. Engineering and Physical Sciences Research Council under grant number GR/K13035 and by British Telecom Labs., Martlesham, Ipswich, U.K.

Processing' we aim to develop tools and techniques that enable the consistency of ODP specifications to be maintained. The multiple viewpoints model is a cornerstone of ODP. In previous papers [DBS95, BDS95] we have investigated unification and consistency in two of the main ODP specification languages: LOTOS [BB88] and Z [Spi89].

For Z, it seems relatively clear what unification entails, because it is a logic-based language with a well understood notion of refinement. A unification of two Z viewpoint specifications is "just" a common refinement of the two, and preferably the "least" common refinement. A previous paper [DBS95] describes a unification algorithm for Z specifications which assumes that names and data types that are to be combined in the unification have already been made almost identical in the viewpoints. We now extend this method by also allowing different descriptions of the same datatype in the viewpoints. This means that relations between the datatypes have to be explicitly formulated. (Such formulations might make up some of the *correspondence rules* described in the ODP Reference Model.) Thus, we tie in unification with notions of data refinement and specification through various *views* [Jac94].

This paper is not written with Z experts in mind – we hope that everyone with a little knowledge of set theory, predicate calculus, and types will be able to follow our expositions. For an introduction to Z, see [BSC94, Rat94, Spi89].

2 Why multiple views of data?

A convincing paper on the use of multiple views of data in Z specifications is one by Daniel Jackson [Jac94], and we will be using part of one of his[1] examples as our running example as well.

Consider the text buffer of an editor, with a cursor somewhere in it. One could imagine various *views* of this entity. We will be presenting two of those: the *File* view and, later, the *Grid* view. Each view has particular operations that are most naturally specified in it, as we will see.

In the *File* view, the buffer is represented by two sequences of some given type *Char*:

File

$left, right : \mathrm{seq}\ Char$

This schema, named *File*, contains two sequences ("seq" is predefined in Z) which represent the text to the left and to the right of the cursor, respectively.

An operation to move the cursor one position to the right can be specified as follows.

[1] The example has a longer history, dating back to [Suf82].

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad csrRight \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Delta File \\
\hline
right \neq \langle\,\rangle \\
left' = left \frown \langle head(right) \rangle \\
right' = tail(right) \\
\hline
\end{array}
$$

By convention, primed components in an operation schema like this one denote the components of the state (*left* and *right*) after the operation. The declaration $\Delta File$ stands for all components of *File* and its primed version $File'$, plus all properties that hold for them according to the *File* schema (i.e., none). The new left string is the concatenation of the old left string with the character directly to the right of the old cursor; the new right string is its tail. There is one precondition for moving the cursor right: it may not be at the end of the buffer, as reflected by the predicate $right \neq \langle\,\rangle$.

Another operation that is easily specified in this view is inserting a character:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad insertChar \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Delta File \\
c? : Char \\
\hline
left' = left \frown \langle c? \rangle \\
right' = right \\
\hline
\end{array}
$$

Input variables of an operation schema are conventionally denoted with question mark decorations.

A completely different view of the same text buffer is the *Grid* view that we will present below. Here, we model the buffer as a sequence of sequences of limited length, assuming a constant *maxlinelength*.

$$
\begin{array}{|l}
maxlinelength : \mathbb{N}
\end{array}
$$

A legal line is one that has a space or a newline as its last character, with no internal newlines.

$$
\begin{array}{|l}
nl, sp : Char \\
\hline
nl \neq sp
\end{array}
$$

$$
\begin{array}{|l}
legallines : \mathbb{P}\,\mathrm{seq}\, Char \\
\hline
\forall\, l : \mathrm{seq}\, Char \bullet \\
\quad l \in legallines \Leftrightarrow (\forall\, i \in \mathrm{dom}\, l \bullet l[i] = nl \;\Rightarrow\; i = \#l) \wedge (l[\#l] \in \{nl, sp\})
\end{array}
$$

Sequences are functions from numbers to some type, so their domains are their index ranges. A sequence of lines is wrapped if it has all legal lines of limited length.

$$\begin{array}{|l} wrapped : \mathbb{P}\,\mathrm{seq}\,\mathrm{seq}\,Char \\ \hline \forall\, ls : \mathrm{seq}\,\mathrm{seq}\,Char \bullet \\ \quad ls \in wrapped \Leftrightarrow (\mathrm{ran}\, ls \subseteq legallines) \land (\forall\, l \in \mathrm{ran}\, ls \bullet \#l \leq maxlinelength) \end{array}$$

The complete *Grid* view of the buffer is then the following.

$$\begin{array}{|l} \textit{Grid} \\ \hline lines : \mathrm{seq}\,\mathrm{seq}\,Char \\ x, y : \mathbb{N} \\ \hline lines \in wrapped \\ y \in \mathrm{dom}\, lines \\ x \in \mathrm{dom}(lines[y]) \\ \hline \end{array}$$

Defining operations like *csrRight* and *insertChar* in this view would be quite a lot of work, since we need to specify what happens "near the end of the line". On the other hand, the following operations cannot be defined on the *File* view: moving the cursor up a line

$$\begin{array}{|l} \textit{csrUp} \\ \hline \Delta Grid \\ \hline y > 1 \land y' = y - 1 \\ x' = min(x, \#lines[y']) \\ lines' = lines \\ \hline \end{array}$$

or deleting until the end of a line

$$\begin{array}{|l} \textit{delEol} \\ \hline \Delta Grid \\ \hline lines'[y] = lines[y][1 \mathrel{..} x] \\ \forall\, z : \mathrm{dom}\, lines \bullet z \neq y \;\Rightarrow\; lines'[z] = lines[z] \\ x' = x \land y' = y \\ \hline \end{array}$$

The invariant relating the two views is that the concatenations of both are equal, and that the cursor is indeed at the end of the *left* sequence. In Jackson's approach, this link is achieved by introducing a schema:

$$\begin{array}{|l} \textit{Editor} \\ \hline File, Grid \\ \hline left \frown right = {\frown}/\, lines \land \#left = x + \Sigma i : 1 \mathrel{..} y - 1 \bullet \#lines[i] \\ \hline \end{array}$$

and then promoting the operations on the individual views to operations on the combined schema – the invariant ensures that the "other" representation is suitably updated. This provides for a nice way of using multiple views for natural specification, each operation can be specified in the view it is most easily expressed in.

There is a small catch, though. The *Grid* view has a few restrictions on its state, and one of these influences the *File* operations as well. Consider what happens if *insertChar* is used to create a word of length *maxlinelength* ... This word can never be part of a wrapped line in the *Grid* view, even though it seemed to cause no problem for *File*. The way of combining views used by Jackson prohibits such use of operations, i.e. one view can actually restrict the applicability of operations defined in another view. Is this the desired effect, or would we rather have a situation where a view sometimes has no current representation because the link between the two views is not "total"? We choose the latter option, as you will see. The most obvious reason for this choice is that we prefer to *weaken*, not strengthen, preconditions when we refine Z specifications.

3 Integrating multiple views into unification

We will recap the simple rule for unification of viewpoints as we presented it in [DBS95].

For unification of data definitions in Z, we assume they have been normalised to the form shown below, where S and T are the maximal types of x in the respective schemas. The strict typing system of Z allows full type checking, which has the advantage of allowing us to talk about the maximal type of an expression, but it has disadvantages as well, as we will see shortly. Our goal is to unify the schemas below.

$$\begin{array}{|l}\hline D \\ x : S \\ \hline pred_S \\ \hline\end{array} \qquad \begin{array}{|l}\hline D \\ x : T \\ \hline pred_T \\ \hline\end{array}$$

We have assumed that normalisation and identification of commonality have already taken place, and identifiers are only identical if they refer to one common object. So, to unify these two schemas we need to find some common type for the two occurrences of x. Informally, this is how it is done:

$$\begin{array}{|l}\hline D \\ x : S \cup T \\ \hline x \in S \Rightarrow pred_S \\ x \in T \Rightarrow pred_T \\ \hline\end{array}$$

However, the rigid type system of Z will insist that $S \cup T$ is a type error, unless S and T happen to be equal. (In practice these types will be equal relatively often because of the normalisation to maximal types. For example with declarations $x : -1 \mathrel{..} 3$ and $y{:}\{z{:}\mathbb{N} \bullet z + z\}$, x and y would both have (maximal) type $\mathbb{Z}$, the restrictions would be added to their schema predicates in normalisation.) Generally, we would have to introduce some kind of disjoint union here, with additional clutter of injections and their inverses. Moreover, if we have the following enumerated types:

$$S = a \mid b \mid c\,,\ T = a \mid d$$

our convention that equal identifiers refer to the same entity implies we want

$$S \cup T = a \mid b \mid c \mid d$$

– which seems increasingly hard to describe in Z. (A disjoint union would include a twice.)

An entirely different argument for *not* taking the unification of two types to be some union is based on the observation that different viewpoints may be on different levels of abstraction. One way in which this variation in abstraction level may show up is in a different view of the datatypes involved. Actually, two viewpoints may even have a different view of the same datatype *without* one of them being more abstract than the other. This implies that we cannot resolve this by performing a data refinement step on the "more abstract" viewpoint first, before applying unification. We have no choice but to stick with both representations in the unified definition, then. We only have to link them – using something very similar to the invariant in the *Editor* schema.

Before we can present the rule for unification, we have to make a small sidestep: we have to introduce "bottom" values for each of the representations, for those cases where only one of the representations is currently valid.

For any type S (note: type, not set), we define the type $S_\perp$ by the following free type definition:

$$S_\perp \ ::=\ \perp_S \mid justS \langle\!\langle S \rangle\!\rangle$$

which states that any value of the type $S_\perp$ is either the constant $\perp_S$, or a value from S labeled with the constructor *justS*. We assume that, for all such types, a function *theS* (a partial surjection) is defined as the inverse of the injection *justS* :

$$\begin{array}{|l}
theS : S_\perp \twoheadrightarrow S \\
\hline
\mathrm{dom}\ theS = \mathrm{ran}\ justS \\
\forall x{:}S \bullet theS\ justS\ x = x
\end{array}$$

and a generic construction for making a partial relation total in both domain and range, by relating all elements that had no image or no original to the correct bottom value:

$$
\begin{array}{|l}
\hline\hline
[S,T] \\
tot : (S \leftrightarrow T) \rightarrowtail\!\!\!\twoheadrightarrow (S_\perp \leftrightarrow T_\perp) \\
\hline
\forall R : S \leftrightarrow T \bullet \\
\quad tot\ R \ = justS \fatsemi R \fatsemi theT \\
\qquad\qquad \cup \{x : S \setminus \mathrm{dom}\ R \bullet (justS\ x, \perp_T)\} \cup \{y : T \setminus \mathrm{ran}\ R \bullet (\perp_S, justT\ y)\} \\
\hline
\end{array}
$$

Here, $\fatsemi$ denotes relational composition, $\times$ is Cartesian product, and *tot* is declared to be a bijection. (End of sidestep.)

In our previous unification rule, the unification between types S and T was rather implicit: if they contained identical values those were assumed to be equal. Since we wish to generalise this, we need to ask explicitly of the specifier to provide a *correspondence* between the two types, stating which value of one type is assumed to represent which value of the other type.

Now, assuming the correspondence for $x{:}S$ and $x{:}T$ is given by the relation $R \subseteq S \times T$, we unify the schemas D given at the beginning of this section to

$$
\begin{array}{|l}
\hline
D \\
x_1 : S_\perp \\
x_2 : T_\perp \\
\hline
(x_1, x_2) \in tot\ R \\
\forall x : S \bullet x_1 = justS\ x \Rightarrow pred_S \\
\forall x : T \bullet x_2 = justT\ x \Rightarrow pred_T \\
\hline
\end{array}
$$

Informally, we maintain two representations of the data. It can be the case that they are both related by the correspondence R, in which case each is required to satisfy its original viewpoint predicate. If one of the representations is outside the (left or right) domain of R, it still satisfies its viewpoint predicate, but the other representation is set to the bottom value in that case.

In the next section we will show how this generalises both Jackson's approach and our simple unification rule.

So much for unifying data type defining schemas – we also need to adapt or possibly unify operation schemas. For an operation that is defined *only* in the first viewpoint by the following schema:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{OpX}} \\
\Delta D \\
\mathit{Decls} \\
\hline
\mathit{pred} \\
\hline
\end{array}
$$

in the unified specification we take

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{OpX}} \\
\Delta D \\
\mathit{Decls} \\
\hline
x_1 \in \operatorname{ran} \mathit{justS} \\
x_1' \in \operatorname{ran} \mathit{justS} \\
\mathbf{let}\ x == \mathit{theS}\ x_1\ ;\ x' == \mathit{theS}\ x_1' \bullet \mathit{pred} \\
\hline
\end{array}
$$

That is, we assert that this operation is defined only if the representation it was defined on is valid. The inclusion of ΔD where D is the unified schema ensures that the other viewpoint representation will change according to the invariant.

If an operation is defined on both viewpoints, we first adapt both schemas to the new unified state as above. Then, to combine them, we make use of the fact that it is possible in Z to compute the pre- and postcondition of an operation and use the rule for operation unification we gave in [DBS95]. Suppose A and B are schemas representing the same operation, both adapted to operate on the same state, they can be unified by the following schema:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{unionAB}} \\
\mathit{Decls} \\
\hline
\operatorname{pre} A \lor \operatorname{pre} B \\
\operatorname{pre} A \Rightarrow \operatorname{post} A \\
\operatorname{pre} B \Rightarrow \operatorname{post} B \\
\hline
\end{array}
$$

where pre and post are the operations that compute pre- and postconditions of a schema. Provided that there are no conflicts between local declarations of both schemas, and no data type unifications besides that for D, *Decls* is obtained by (textually) unifying the declarations of both schemas.

A proof that these rules result in a (most general) common refinement, with the consistency conditions on operations that are needed to guarantee this, can be found in a forthcoming paper [BDBS95].

4 Examples

Three cases for S and T can be distinguished in our previous rule: they are identical, they are disjoint, or they have some overlap. (The latter two cases require an explicit new datatype definition to satisfy Z's typing restrictions.) We show in which ways these are all instances of the new rule.

- When S and T are equal, we take R to be the identity relation on S, i.e. $\{x{:}S \bullet (x,x)\}$. Since the identity relation is a total relation, $tot\ R{=}\{x{:}S \bullet (justS\ x, justS\ x)\}$. The resulting schema is equivalent, by the isomorphism[2] $(justS\ x, justS\ x) \leftrightarrow x$, to:

$$\begin{array}{|l} \multicolumn{1}{l}{D} \\ \hline x{:}S \\ \hline pred_S \\ pred_T \\ \hline \end{array}$$

as expected. For example, the schemas

$$\begin{array}{|l} \multicolumn{1}{l}{D} \\ \hline x : -1 \mathinner{..} 3 \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{D} \\ \hline x{:}\{z{:}\mathbb{N} \bullet z + z\} \\ \hline \end{array}$$

would get normalised to

$$\begin{array}{|l} \multicolumn{1}{l}{D} \\ \hline x{:}\mathbb{Z} \\ \hline -1 \leq x \leq 3 \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{D} \\ \hline x{:}\mathbb{Z} \\ \hline \exists\, z{:}\mathbb{N} \bullet x = z + z \\ \hline \end{array}$$

and their unification with $R{=}Z \times Z$ would essentially be their intersection:

$$\begin{array}{|l} \multicolumn{1}{l}{D} \\ \hline x{:}\mathbb{Z} \\ \hline -1 \leq x \leq 3 \\ \exists\, z{:}\mathbb{N} \bullet x = z + z \\ \hline \end{array}$$

i.e. $x{=}0$ or $x{=}2$. If we wanted the union or disjoint union of these sets, we would take different R.

[2]Isomorphisms in this context are just renamings of the inhabitants of a schema, i.e. total and one-to-one data refinements in both directions.

- When S and T are disjoint, they are in no way related, and thus we need to take $R{=}\emptyset$. Then *tot* R only contains pairs of the forms $(justS\ x, \perp_T)$ and $(\perp_S, justT\ y)$. This is actually a different representation of the disjoint union of S and T. We would define this as

  $$uniST \ ::= \ inS\langle\!\langle S\rangle\!\rangle \mid inT\langle\!\langle T\rangle\!\rangle$$

  By the isomorphism $(justS\ x, \perp_T) \leftrightarrow inS\ x$, $(\perp_S, justT\ y) \leftrightarrow inT\ y$, the schema resulting from unification is equivalent to:

  $$\begin{array}{|l}
  \multicolumn{1}{l}{D} \\ \hline
  xx : uniST \\ \hline
  \forall x{:}S \bullet xx{=}inS\ x \Rightarrow pred_S \\
  \forall x{:}T \bullet xx{=}inT\ x \Rightarrow pred_T \\ \hline
  \end{array}$$

  as we would expect. For example, if $S = a \mid b$ and $T = c \mid d$, the unified schema would contain the pairs $(justS\ a, \perp_T)$, $(justS\ b, \perp_T)$, $(\perp_S, justT\ c)$ and $(\perp_S, justT\ d)$ satisfying the relevant predicates, and so we might as well take the isomorphic schema:

  $$\begin{array}{|l}
  \multicolumn{1}{l}{D} \\ \hline
  x : a \mid b \mid c \mid d \\ \hline
  x \in \{a, b\} \Rightarrow pred_S \\
  x \in \{c, d\} \Rightarrow pred_T \\ \hline
  \end{array}$$

- For the case that S and T overlap, we will look at the concrete example used earlier: $S = a \mid b \mid c$, $T = a \mid d$. If we take R to be $\{(a,a)\}$, *tot* R is

  $$\{(justS\ a, justT\ a), (justS\ b, \perp_T), (justS\ c, \perp_T), (\perp_S, justT\ d)\}$$

  which is obviously isomorphic to the type $a \mid b \mid c \mid d$.

These were all examples of the new rule resulting in schemas equivalent to the ones produced by the old rule, and in all cases R was a subset of the identity relation.

For applying the new rule to the *File*/*Grid* example we need to adapt the schemas in the specification slightly. The rule is defined on single components only, whereas we need to relate, on the one hand, components *left* and *right* to, on the other hand, components *lines*, x and y. One way of doing this would be to redefine *File* and *Grid* to have single tupled components, introducing a clutter of extra names and projections. However, Z's schema calculus allows us to use the name of a schema as a record type, with the components as the fields, restricted to the records satisfying the predicate. We just have to add two dummy schemas:

$$\begin{array}{|l}\hline \textit{FileFile} \\ x : \textit{File} \\ \hline\end{array} \qquad \begin{array}{|l}\hline \textit{GridGrid} \\ x : \textit{Grid} \\ \hline\end{array}$$

and unify these two. The correspondence between the types *File* and *Grid* is then expressed by the relation R defined by

$$\begin{array}{|l} R : \textit{File} \leftrightarrow \textit{Grid} \\ \hline \forall f : \textit{File};\ g : \textit{Grid} \bullet (f, g) \in R \ \Leftrightarrow\ f.\textit{left} \frown f.\textit{right} = \frown /\ g.\textit{lines} \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \# f.\textit{left} = g.x + \Sigma i : 1 \,..\, g.y - 1 \bullet \# g.\textit{lines}[i] \end{array}$$

Because the predicates for *FileFile* and *GridGrid* are true, their unification with R is the simple schema

$$\begin{array}{|l}\hline \textit{FileandGrid} \\ x_1 : \textit{File}_\perp \\ x_2 : \textit{Grid}_\perp \\ \hline (x_1, x_2) \in \textit{tot}\ R \\ \hline\end{array}$$

We observed earlier that every *Grid* can be represented as a *File*, and vice versa iff there is no word longer than *maxlinelength*. Thus, the above schema is equivalent to

$$\begin{array}{|l}\hline \textit{FileandGrid} \\ x_1 : \textit{File} \\ x_2 : \textit{Grid}_\perp \\ \hline (x_1, \textit{theGrid}\ x_2) \in R \vee (x_2 = \perp_{\textit{Grid}} \wedge x_1 \in \textit{Fileswithwordslongerthanmax}) \\ \hline\end{array}$$

for a suitably defined set.

This schema, in turn, can serve as an implementation of the *Editor* schema resulting from Jackson's method. If we strengthen the predicate by removing the second disjunct, and then "unpack" the record types *File* and *Grid*, we get the *Editor* schema.

We observed that taking R to be a partial identity relation reduces the new rule to the rule we presented earlier [DBS95]. It is also clear that taking R to be a total relation reduces the new rule to Jackson's method. Without a limit on the length of lines, both unified schemas would be identical. The difference is caused by the invariant relation being partial in one of its domains: Jackson reduces the domain to make the invariant total again, we extend the relation using bottoms, making it "total" on a larger domain.

5 Conclusions and further research

We have succeeded in integrating previously published methods for unifying Z specifications using different views and viewpoints. Thus, we allow an element of data type refinement or translation to be included in unification. This has the advantage that, whenever different datatype representations of one entity are used in a modular specification effort, we do not need to choose one representation over another at an early stage. Several representations can coexist, and the specifications they are used in can be checked for consistency using unification.

A technical problem with this approach is the abundance of bottom values and newly defined free types with all their constructor functions and their inverses. It would be nice if we could do away with some of these, and we will be investigating this topic.

Another issue that we would like to explore is the relevance of this generalisation for viewpoint specifications in the RM-ODP framework. Is it helpful to allow multiple representations of the same entity across viewpoints, in particular for the viewpoints that Z will be used in? Will some of the RM-ODP correspondence rules represent data type transformations? We aim to apply our techniques to RM-ODP viewpoint specifications to answer such questions, and thus to check the usability of these methods for ODP specification.

References

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.

[BDBS95] E. Boiten, J. Derrick, H. Bowman, and M. Steen. Consistency and refinement for partial specification in Z. Submitted for publication, 1995.

[BDS95] H. Bowman, J. Derrick, and M. Steen. Some results on cross viewpoint consistency checking. In *IFIP International Conference on Open Distributed Processing*. Chapman & Hall, 1995.

[BSC94] R. Barden, S. Stepney, and D. Cooper. *Z in practice*. Prentice Hall, 1994.

[DBS95] J. Derrick, H. Bowman, and M. Steen. Maintaining cross viewpoint consistency using Z. In *IFIP International Conference on Open Distributed Processing*. Chapman & Hall, 1995.

[FGH+94] A.C.W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.

[ITU95] ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.

[Jac94] D. Jackson. Structuring Z specifications with views. Technical Report CMU-CS-94-126, School of Computer Science, Carnegie Mellon University, 1994.

[Rat94] B. Ratcliff. *Introducing specification using Z.* McGraw-Hill, 1994.

[Spi89] J.M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1989.

[Suf82] B.A. Sufrin. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1:157–202, 1982.

[ZJ93] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.

This paper was typeset in LaTeX using the MathPad (http://www.win.tue.nl/win/cs/wp/mathspad/) editing tool with specially written stencils for using oz.sty.

To appear in the proceedings of Computing Science in the Netherlands, November 1995.