Computer Science
University of Manchester

Understanding the differences between VDM and Z

I. J. Hayes, C. B. Jones and J. E. Nicholls

Technical Report UMCS–93–8–1

Understanding the differences between VDM and Z

I. J. Hayes*

Department of Computer Science
University of Queensland
e-mail: Ian.Hayes@uqcspe.cs.uq.oz.au

C. B. Jones

Department of Computer Science
University of Manchester
e-mail: cbj@cs.man.ac.uk

J. E. Nicholls

Programming Research Group
Oxford University

August 11, 1993

Abstract

This paper attempts to provide an understanding of the interesting differences between two well-known specification languages.

Technical Reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `ftp.cs.man.ac.uk` in the directory `/pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, U.K.

The main ideas are presented in the form of a discussion. This was partly prompted by Lakatos' book 'Proof and Refutations' but, since this paper is less profound, characters from the childrens' television series 'The Magic Roundabout' are the speakers: Zebedee speaks for Z, Dougal puts the VDM position, and Florence acts as the user.

The specifications which are presented have been made similar so as to afford comparison – in neither the VDM nor the Z case would they be considered to be ideal presentations. Some technical details are relegated to footnotes.

Discussion

Florence: I know that some people are confused by the existence of two specification languages, Z and VDM, which have a lot in common.

Dougal: Yes, there is certainly a common objective in Z and VDM and in many places one can see that the same solution has been adopted.

The use of both VDM and Z has been concentrated on the specification of *abstract machines*, and they both take the same so called 'model-oriented' approach. Jointly they differ from the so called 'algebraic' specification languages (these might be better called 'property-oriented' to contrast with 'model-oriented') which concentrate on specifying abstract data types.

Zebedee: The primary difference between these approaches is that VDM and Z both give an explicit model of the state of an abstract machine – the operations of the abstract machine are defined in terms of this state – whereas 'algebraic' approaches give no explicit model of the type – an abstract data type is specified by axioms giving relationships between its operations. For (a much-overused) example, a stack in VDM and Z would typically be modelled as a sequence, while in the 'algebraic' approaches axioms such as

$$pop(push(x, s)) = s$$

would be given.

Dougal: Before going further, let's be clear what we mean by 'VDM'. Strictly, VDM was always seen as a development method in the sense that rules are given to verify steps of development (data reification and operation decomposition). I guess that our discussion today will be confined to the specification language used in VDM.

Florence: Does that have a name?

Dougal: I wish I could say 'no'! But I have to confess that it is sometimes known as 'Meta-IV' – the draft VDM standard talks about 'VDM-SL',[1] but let's talk about the significant differences between the two specification languages.

Florence: Probably what hit me first about the difference between VDM and Z specifications is the appearance of the page. VDM specifications are full of keywords and Z specifications are all 'boxes'. Is this a significant difference?

Dougal: VDM uses keywords in order to distinguish the roles of different parts of a specification. For example, a module corresponds to an abstract machine with state and operations; a state has components and an invariant; and operations have separate pre- and post-conditions. These

[1] The notation used here is close to that of the 'Committee Draft' of the VDM-SL Standard but there may be minor syntactic differences.

different components are distinguished as they have different purposes in the specification. To do this VDM makes use of keywords to introduce each component. All this structure is part of the VDM language.

Zebedee: In Z almost none of the structure Dougal mentioned is explicit in the language. A typical specification consists of lots of definitions, many of which are *schemas* – the boxes Florence was referring to. Schemas are used to describe not only states but also operations. The status of any particular schema is really only determined by the text introducing it, although it isn't hard to guess the purpose of a schema by looking at its definition. Schemas are also used as building blocks to construct descriptions of machine states or facets of operations. Such component building blocks may not correspond to any of the VDM categories.[2]

Florence: But don't you need the extra structure that VDM gives if you want to do formal refinements?

Zebedee: Yes, but to perform refinements you also need to consider a target programming language. For example, if you wanted to produce Modula code then you could give a definition module for an abstract machine in which the state and the operations are defined by Z schemas. Such a definition module would be very close to a VDM module in terms of structure.

Florence: How about an example – something other than stacks please!

Zebedee: A simple relational database, known as NDB, has been presented in both notations. Let's use that.[3] After you Dougal.

Dougal: Someone writing a VDM specification of a system normally sketches a state before going into the details of the operations which use and modify that state.[4]

For the NDB description, the overall state will have information about entities and relations. In order to build up such a state, a number of sets are taken as basic.

Eid Each entity in the database has a unique identifier taken from the set *Eid*.

Value Entities have values taken from the set *Value*.

Esetnm An entity can belong to one or more entity sets (or types). The names of these sets are taken from the set *Esetnm*.

Rnm The database consists of a number of relations with names taken from the set *Rnm*.

All but the first of these can conveniently be thought of as parameters to the specification of the NDB database system; *Eid* is an internal set about which we need to know little – we just regard it as a set of 'tokens'.

Now we can begin to think about the types which are constructed from these basic sets. NDB is a binary relational database consisting of relations containing tuples which each have

[2] It is perhaps worth explaining that Z has a number of levels: a basic mathematical notation (similar to that of VDM); the schema notation; and conventions for describing the state and operations of an abstract machine using Z schemas. Little new notation is introduced in the third level, only conventions for making use of the other notation to describe abstract machines. It is worth noting that Z notation – the first two levels – has been used with a different set of conventions for other purposes, such as specifying real-time systems.

[3] Originally formally specified in [Wel82] and revised in [Wal90], this was used as a challenge problem in [FJ90] to which a Z response is given in [Hay92].

[4] The description of the NDB state presented here is given *postfacto* rather than attempting to emulate the process by which specifications are produced.

two elements: a *from* value and a *to* value. A tuple contains a pair of *Eid*s; in VDM the type *Tuple* is defined as follows

$$\begin{aligned} \mathit{Tuple} :: \ & \mathit{fv} : \mathit{Eid} \\ & \mathit{tv} : \mathit{Eid} \end{aligned}$$

Then a relation can be defined as a set of such pairs:

$$\mathit{Relation} = \mathit{Tuple}\text{-}\mathsf{set}$$

To define whether a relation is to be constrained to be one-to-one – or whatever – four distinct constants are used

$$\mathit{Maptp} = \textsc{OneOne} \mid \textsc{OneMany} \mid \textsc{ManyOne} \mid \textsc{ManyMany}$$

Relation information (*Rinf*) contains information stored for a relation: apart from the *Tuple* set, an element of *Maptp* provides the constraint on the form of relation allowed. The consistency of the *tp* and *r* fields of *Rinf* is expressed as an invariant.

$$\begin{aligned} \mathit{Rinf} :: \ & \mathit{tp} : \mathit{Maptp} \\ & r \ \ : \mathit{Relation} \end{aligned}$$

$$\begin{aligned} & \mathsf{inv}\ (\mathit{mk}\text{-}\mathit{Rinf}(\mathit{tp}, r)) \triangleq \\ & \quad (\mathit{tp} = \textsc{OneMany} \ \Rightarrow\ \forall t_1, t_2 \in r \cdot t_1.\mathit{tv} = t_2.\mathit{tv} \ \Rightarrow\ t_1.\mathit{fv} = t_2.\mathit{fv}) \wedge \\ & \quad (\mathit{tp} = \textsc{ManyOne} \ \Rightarrow\ \forall t_1, t_2 \in r \cdot t_1.\mathit{fv} = t_2.\mathit{fv} \ \Rightarrow\ t_1.\mathit{tv} = t_2.\mathit{tv}) \wedge \\ & \quad (\mathit{tp} = \textsc{OneOne} \ \Rightarrow\ \forall t_1, t_2 \in r \cdot t_1.\mathit{fv} = t_2.\mathit{fv} \ \Leftrightarrow\ t_1.\mathit{tv} = t_2.\mathit{tv}) \end{aligned}$$

It's worth noting that the set of values defined by a definition with an invariant only contains values which satisfy the invariant. So we can only say that $\mathit{rel} \in \mathit{Rinf}$ if the relation, $\mathit{rel}.r$ is consistent with the map type $\mathit{rel}.\mathit{tp}$. Because invariants can be arbitrary predicates, type membership is only partially decidable.

Florence: Is Z's notion of type the same as what Dougal just described?

Zebedee: No, but this is a difference in the use of the word 'type' rather than a real difference between Z and VDM. In Z the term 'type' is used to refer to what can be statically type checked. This is more liberal than what Z calls a 'declared set' which is what VDM calls a 'type'.

Dougal: Well, let me get through the rest of the state; then we can make more comparisons.

In NDB, relations are identified not only by their names but also by the entity sets of the values they relate (so a database can contain two relations called OWNS: one between PEOPLE and CARS and – at the same time – another between PEOPLE and HOUSES). So a key for a relation contains three things

$$\begin{aligned} \mathit{Rkey} :: \ & \mathit{nm} : \mathit{Rnm} \\ & \mathit{fs} \ \ : \mathit{Esetnm} \\ & \mathit{ts} \ \ : \mathit{Esetnm} \end{aligned}$$

The overall state which we are aiming for has three components: an entity set map (*esm*) which defines which entities are in each valid entity set; a map (*em*) which contains the value of each identified entity; and a third map (*rm*) which stores the relevant *Rinf* for each *Rkey*. The invariant records the consistency conditions between the components.

$$\begin{aligned} \mathit{Ndb} :: \ & \mathit{esm} : \mathit{Esetnm} \xrightarrow{m} \mathit{Eid}\text{-}\mathsf{set} \\ & \mathit{em} \ \ : \mathit{Eid} \xrightarrow{m} \mathit{Value} \\ & \mathit{rm} \ \ : \mathit{Rkey} \xrightarrow{m} \mathit{Rinf} \end{aligned}$$

$\mathsf{inv}\ (mk\text{-}Ndb(esm, em, rm)) \triangleq$
$\quad \mathsf{dom}\, em = \bigcup \mathsf{rng}\, esm \wedge$
$\quad \forall rk \in \mathsf{dom}\, rm \cdot$
$\qquad \{rk.fs, rk.ts\} \subseteq \mathsf{dom}\, esm \wedge$
$\qquad \forall mk\text{-}Tuple(fv, tv) \in rm(rk).r \cdot fv \in esm(fs) \wedge tv \in esm(ts)$

Later, we'll look at how this (together with the initial state and the operations) gets grouped into a module.

Florence: How would the above state be presented in Z?

Zebedee: All the details would be almost identical, but the specification would be structured differently. The specification would consist of a sequence of sections and each section would present a small set of the state components along with operations on just those components.

The Z approach to structuring specifications is to try to build the specification from near orthogonal components. We look for ways of splitting the state of the system so that we can specify operations on just that part of the state that they require.

For NDB we have chosen to split the specification into three parts:

1. entities and their types or entity sets,
2. a single relation, and
3. multiple relations.

Finally, we put these specifications together to give the final specification.

Rather than follow the normal Z approach here, I'll give all of the state components from the different sections together, so that we can compare the state with that used for the VDM specification. As for the VDM, our basic sets are the following:

$[Eid, Esetnm, Value]$

As with the VDM the sets *Esetnm* and *Value* can be thought of as parameters, and the set *Eid* is used locally within the specification.

For a database we keep track of the entities that are in an entity set (of that type). Every entity must be of one or more known types. The following schema, *Entities*, groups the components of the state together with a invariant linking them.

Entities

$esm: Esetnm \mathrel{+\!\!\!\!\twoheadrightarrow\!\!\!\!\!\!\!\!+} (\mathbb{F}\ Eid)$
$em: Eid \mathrel{+\!\!\!\!\twoheadrightarrow\!\!\!\!\!\!\!\!+} Value$

$\mathrm{dom}\, em = \bigcup \mathrm{ran}\, esm$

Florence: How does this differ from the VDM so far?

Zebedee: It's virtually identical – bar the concrete syntax. The notation $\mathbb{F}$ *Esetnm* is equivalent to the VDM notation Esetnm-set.

Another approach to defining the state in Z would be to define *esm* as a binary relation between *Esetnm* and *Eid*. This leads to simpler predicates in the specifications because the Z operators on binary relations are closer to the operations required for NDB's binary relations. *Entities* is defined using Z's binary relations in [Hay92], but for our comparison here it is simpler to use the same state as the VDM version. That way we can concentrate on more fundamental differences.

Dougal: Yes, these modelling differences are interesting but not the point of today's discussion; but it is worth saying that binary relation notation could also be added to VDM and the same alternative state used.

Florence: So far that only covers the components *esm* and *em* of the VDM *Ndb* state.

Zebedee: Right, and at this point we would give a set of operations on the above state, but in order to provide a more straightforward comparison with the VDM state, we shall skip to the remainder of the state. The relations used in NDB are binary relations between entity identifiers (rather than entity values). A *Tuple* is just a pair of entity identifiers and a *Relation* is modeled as a Z binary relation between entity identifiers.

$$\begin{aligned} &\mathit{Tuple} == \mathit{Eid} \times \mathit{Eid} \\ &\mathit{Relation} == \mathit{Eid} \leftrightarrow \mathit{Eid} \end{aligned}$$

Florence: Is that really different from VDM?

Zebedee: Well, yes and no. Both the VDM and Z versions use a set of pairs for a relation – note that $\mathit{Eid} \leftrightarrow \mathit{Eid}$ is a shorthand for $\mathbb{P}(\mathit{Eid} \times \mathit{Eid})$. The difference is that, in Z, relations are predefined and have a rich set of operators defined on them.

Dougal: There are a couple of points I'd like to pick up from what Zebedee has said. When we make a selection of basic building blocks for specifications, we are clearly influenced by experience. There is nothing deep in, say, the omission of relations from VDM (or – say of optional objects from Z). Once again, the remarkable thing is just how similar the selection in Z and VDM is. As I said above, if I were writing a large specification which needed relations, I would just extend VDM appropriately.

Florence: What about the *Maptp* in Z?

Zebedee: *Maptp* is virtually identical:

$$\mathit{Maptp} ::= \mathit{OneOne} \mid \mathit{OneMany} \mid \mathit{ManyOne} \mid \mathit{ManyMany}$$

When a relation is created its type is specified as being one of the following four possibilities: it is a one-to-one relation (i.e., an injective partial function), a one-to-many relation (i.e., its inverse is a partial function), a many-to-one relation (i.e., a partial function), or a many-to-many relation. In Z, the set of binary relations between X and Y is written $X \leftrightarrow Y$, the set of partial functions is written $X \rightarrow\!\!\!\!\!+ Y$, the set of one-to-one partial functions is written $X \rightarrowtail\!\!\!\!\!+ Y$, and the inverse of a relation r is written $r^\sim$.

A relation is created to be of a particular type and no operation on the relation may violate the type constraint.

$$\begin{array}{|l} \hline \mathit{Rinf} \\ \mathit{tp}: \mathit{Maptp} \\ r: \mathit{Relation} \\ \hline (\mathit{tp} = \mathit{OneOne} \Rightarrow r \in \mathit{Eid} \rightarrowtail\!\!\!\!\!+ \mathit{Eid}) \land \\ (\mathit{tp} = \mathit{ManyOne} \Rightarrow r \in \mathit{Eid} \rightarrow\!\!\!\!\!+ \mathit{Eid}) \land \\ (\mathit{tp} = \mathit{OneMany} \Rightarrow r^\sim \in \mathit{Eid} \rightarrow\!\!\!\!\!+ \mathit{Eid}) \\ \hline \end{array}$$

Dougal: If you expanded the definitions you would get the same constraint as in the VDM version.

Zebedee: Yes, they are exactly the same.

Florence: We still don't have NDB's named relations in Z.

Zebedee: That's next, but again at this point in the normal flow of a Z specification operations would be defined on the state *Rinf*. The database consists of a number of relations with names taken from the set *Rnm* (essentially a parameter set).

$$[Rnm]$$

A relation is identified by its name and the 'from' and 'to' entity sets that it relates. This allows a number of relations to have the same *Rnm* provided they have different combinations of 'from' and 'to' entity sets.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Rkey}} \\
nm : Rnm \\
fs, ts : Esetnm \\
\hline
\end{array}$$

The entities related by each relation must belong to the entity sets specified by the relation key.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Ndb}} \\
Entities \\
rm : Rkey \mathrel{\ooalign{$\rightarrow$\cr\hfil$\mid\mid$\hfil}} Rinf \\
\hline
\forall rk : \mathrm{dom}\ rm \bullet \\
\quad \{rk.fs, rk.ts\} \subseteq \mathrm{dom}\ esm \land \\
\quad (\forall t : (rm\ rk).r \bullet \\
\qquad first\ t \in esm(rk.fs) \land second\ t \in esm(rk.ts)) \\
\hline
\end{array}$$

Florence: Because you have included *Entities* you now have all the state components, so that this is equivalent to the VDM *Ndb* state.

Zebedee: Yes.

Florence: Why don't we look at initialisation?

Dougal: The initial state (in this case it is unique) is defined in VDM as

$$\mathsf{init}\,(ndb) \quad \triangleq \quad ndb = mk\text{-}Ndb(\{\,\}, \{\,\}, \{\,\})$$

Zebedee: In Z the initial state is defined by the following schema (given using the horizontal form of presentation):

$$Ndb\_Init \mathrel{\widehat{=}} [Ndb \mid esm = \{\} \land em = \{\} \land rm = \{\}]$$

Again, if we followed the more structured presentation of NDB, we would probably define the set of allowable initial *Entities* and then define the allowable initial Ndb states in terms of it.

Florence: Why don't we look at operations?

Dougal: The simplest operation adds a new entity set name to the set of known entity sets, esm. The set of entities associated with this new name is initially empty. In VDM this can be defined.

$$
\begin{array}{l}
ADDES\ (es\colon Esetnm) \\
\mathsf{ext\ wr}\ esm\ :\ Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set} \\
\mathsf{pre}\ es \notin \mathsf{dom}\, esm \\
\mathsf{post}\ esm = \overleftarrow{esm} \cup \{es \mapsto \{\,\}\}
\end{array}
$$

Zebedee: In Z that's

$$
\begin{array}{|l}
\hline
ADDES0 \\
\Delta Entities \\
es?\colon Esetnm \\
\hline
es? \notin \mathrm{dom}\, esm\ \wedge \\
esm' = esm \cup \{es \mapsto \{\}\}\ \wedge \\
em' = em \\
\hline
\end{array}
$$

where $\Delta Entities$ introduces the *before* and *after* (primed) states:

$$\Delta Entities \mathrel{\widehat=} Entities \wedge Entities'$$

Florence: One obvious syntactic difference is that VDM uses hooked variables for the *before* state and unhooked variables for the *after* state, whilst Z uses undecorated variables for the *before* state and primed variables for the *after* state. In addition, Z uses variable names ending in '?' for inputs (and names ending in '!' for outputs). Apart from the differences in syntax, I notice that VDM uses an externals clause and distinguishes the pre-condition.

Zebedee: Yes, Z does not have any equivalent of an externals clause. The predicate must define the final values of all variables, even if the variable is unchanged, such as em. This is why in Z one divides up the state into small groups of components and defines sub-operations on each group, before combining the sub-operations in order to define the full operation. For a large specification with many state components, if one had to define the operations on the whole state, then there would be many boring predicates stating that many of the variables are unaffected. Dividing up the state avoids this problem, although it is still necessary to promote the operations on the substates to the full state at some stage.

Dougal: In VDM, an operation is always written in a module. This provides the appropriate state but one can use the external clause of an operation specification to make it self-contained and to restrict the frame.

Zebedee: In Z, the state is explicitly included – via a 'Δ' or 'Ξ' schema usually – within the operation, so the operation schema can stand on its own.

With regard to the pre-condition, although the same logical expression appears in the Z schema $ADDES0$ as in the VDM operation $ADDES$, it is not separated out. For this operation that doesn't make a large difference between the Z and VDM versions, but for other operations it can.

Dougal: OK, let's look at deleting an entity set in VDM.

$DELES\ (es: Esetnm)$

$\mathsf{ext\ wr}\ esm\ :\ Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}$

$\quad \mathsf{rd}\ rm\ \ :\ Rkey \xrightarrow{m} Rinf$

$\mathsf{pre}\ es \in \mathsf{dom}\, esm \wedge esm(es) = \{\,\} \wedge$

$\quad \forall rk \in \mathsf{dom}\, rm \cdot es \neq rk.fs \wedge es \neq rk.ts$

$\mathsf{post}\ esm = \{es\} \lhd \overleftarrow{esm}$

Zebedee: In Z this would be written

$$\begin{array}{|l}
\hline DELES0 \\
\Delta Entities \\
es?: Esetnm \\
\hline
es? \in \mathrm{dom}\, esm \wedge esm(es) = \{\} \wedge \\
esm' = \{es?\} \lhd esm \wedge \\
em' = em \\
\hline
\end{array}$$

Florence: But isn't it missing part of the pre-condition in the VDM version?

Zebedee: Yes, $DELES0$ is only defined on the state $Entities$, so it is impossible to talk about the state component rm. To define the equivalent of the VDM operation, we need to promote $DELES0$ to the full Ndb state. We do this by defining the schema ΞRM which introduces the full Ndb state and constrains rm to be unchanged. This is then conjoined with $DELES0$.

$$\Xi RM \mathrel{\widehat{=}} [\Delta Ndb \mid rm' = rm]$$
$$DELES \mathrel{\widehat{=}} DELES0 \wedge \Xi RM$$

Florence: But I still can't see the missing bit of the pre-condition!

Zebedee: That's because it's not visible! But the Z $DELES$ has the same pre-condition as the VDM operation.

Florence: How do I get to see it?

Zebedee: In Z, the pre-condition of an operation characterises exactly those inputs and initial states such that there exists a least one possible combination of outputs and final state that satisfies the operation specification. For $DELES$ the pre-condition is

$$\begin{array}{|l}
\hline \mathrm{pre}\ DELES \\
Ndb \\
es?: Esetnm \\
\hline
\exists\, Ndb' \bullet \\
\qquad es? \in \mathrm{dom}\, esm \wedge esm(es) = \{\} \wedge \\
\qquad esm' = \{es?\} \lhd esm \wedge \\
\qquad rm' = rm \\
\hline
\end{array}$$

The predicate can be expanded to

$$
\begin{array}{|l}
\hline
\text{pre } \mathit{DELES} \\
\mathit{Ndb} \\
es?: \mathit{Esetnm} \\
\hline
\exists \mathit{Entities}'; rm': \mathit{Rkey} \pfun \mathit{Rinf} \bullet \\
\quad (\forall rk: \mathrm{dom}\, rm' \bullet \\
\qquad \{rk.fs, rk.ts\} \subseteq \mathrm{dom}\, esm' \wedge \\
\qquad (\forall t: (rm'\, rk).r \bullet \\
\qquad\quad \mathit{first}\, t \in esm'(rk.fs) \wedge \mathit{second}\, t \in esm'(rk.ts))) \wedge \\
\quad es? \in \mathrm{dom}\, esm \wedge esm(es) = \{\} \wedge \\
\quad esm' = \{es?\} \ndres esm \wedge \\
\quad rm' = rm \\
\hline
\end{array}
$$

which can be simplified to

$$
\begin{array}{|l}
\hline
\text{pre } \mathit{DELES} \\
\mathit{Ndb} \\
es?: \mathit{Esetnm} \\
\hline
es? \in \mathrm{dom}\, esm \wedge esm(es) = \{\} \wedge \\
(\forall rk: \mathrm{dom}\, rm \bullet es? \neq rk.fs \wedge es? \neq rk.ts) \\
\hline
\end{array}
$$

Dougal: That's now the same as the VDM pre-condition.

Florence: Wasn't all that a bit complicated compared to the VDM version?

Zebedee: Well, yes and no. If you want to compare it with the VDM version, then we have also done the equivalent of discharging its *satisfiability* proof obligation. In VDM this proof obligation is the main consistency check available for the specifier, and the Z pre-condition calculation can be likewise seen as a consistency check – that the calculated pre-condition agrees with the specifier's expectations.

Dougal: I believe that this *is* a significant difference between Z and VDM. There is a technical point: when development steps are undertaken, the pre-condition is required.[5] But there is also a pragmatic point: in reading many industrial (informal) specifications, I have observed that people are actually not so bad at describing what function is to be performed; what they so often forget is to record the assumptions. I therefore think that it is wise to prompt a specifier to think about the pre-condition.

Zebedee: I'd agree with that, but in both VDM and Z, provided the respective consistency checks are done, we do end up at the same point. It's only the path that is different.

With the Z approach of constructing the specification of an operation it is only when you have the final operation that the concept of a pre-condition really makes sense.[6]

Florence: What does a pre-condition mean? I'm really not clear about whose responsibility it is to avoid calls that violate the pre-condition – or are these exceptions?

[5] Tony Hoare in [Hoa91] appears to argue for the use of Z to develop specifications and the use of VDM for development of the design and implementation.

[6] Although it is possible to define operators similar to schema conjunction and disjunction on pre/postcondition pairs; see [War93].

Dougal: A pre-condition is essentially a warning to the user: the behaviour defined in the post-condition is only guaranteed if the starting state satisfies the pre-condition. In formal development, the user should treat the pre-condition as a proof obligation that an operation is only invoked when its pre-condition is true. It is perhaps useful to think of the pre-condition as something that the *developer* of an operation can rely upon; it is permission for the developer to ignore certain situations. Exceptions are quite different – VDM does have notation (not used so far in this example) to mark error conditions which must be detected and handled by the developer. In VDM an operation specification can consist of a normal pre/postcondition pair followed by a set of exception pre/postcondition pairs.

Florence: Can we compare the treatment of exceptions? I've noticed Z doesn't have exceptions as part of the language.

Zebedee: I guess one way of viewing the Z approach is to say that it doesn't really have exceptions at all. As part of specifying an operation one specifies its behaviour both in the normal case and in the exceptional cases. Both the normal case and each exceptional case are specified in the same manner in a Z schema. Each of these schemas corresponds to a pre/post-condition pair in the VDM version.

Florence: So the difference about the use of pre/postcondition pairs in the VDM version versus a single schema in the Z version crops up here as well.

Zebedee: That's correct.

Florence: How else do they differ?

Zebedee: In terms of what they both mean, not at all. The complete operation is specified in Z by taking the disjunction of the schemas for the normal and exceptional cases. It has the same meaning as the corresponding VDM specification. For example, in both Z and VDM the preconditions of the alternatives may overlap, either between the normal case and an error case or between error alternatives, and in both Z and VDM there is a nondeterministic choice between alternatives that overlap.

Dougal: Yes, that's correct. Although VDM and Z appear to describe exceptions differently, the semantic ideas underneath the concrete syntax are virtually identical.

Florence: So why do they look so different?

Zebedee: Well mostly it is just differences in syntax but I guess there are a couple of points about building an operation specification from Z schemas using schema operators that are worth noting. Firstly, it is possible to specify more than one normal case and these further alternatives just become part of the disjunction of cases. (There is no real distinction between normal and error cases, so one can have as many of either as suits the problem in hand.) Secondly, the same *exception* schema can be used for more than one operation. This has the twin advantages of avoiding repetition and maintaining consistency between different operations in their treatment of the same exception.

Dougal: I see those advantages; it is just in the spirit of VDM to have a place for exceptions marked by keywords. As always, syntactic issues tend to be more an issue of taste than of hard scientific arguments.

Florence: How about an example?

Zebedee: Let's consider the possibility of trying to add a new entity set when the name is already in use. In Z the exception alternative is specified by

$$\begin{array}{|l}\hline \multicolumn{1}{l}{ESInUse} \\ \Xi Entities \\ es?: Esetnm \\ \hline es? \in \operatorname{dom} esm \\ \hline\end{array}$$

where $\Xi Entities$ introduces the *before* and *after* states and constrains them to be equal:

$$\Xi Entities \mathrel{\widehat=} [\Delta Entities \mid \theta Entities' = \theta Entities]$$

The operation augmented with the error alternative is

$$ADDESX \mathrel{\widehat=} ADDES0 \lor ESInUse$$

Dougal: In VDM *ADDES* with an exception if the entity set is already in use would be specified by adding the line

$\mathsf{err}\ \text{ESI}\textsc{n}\textsc{Use}\ \ es \in \mathsf{dom}\, esm$

Zebedee: In both the VDM and Z we should really add an error report to be returned by the operation. This would be done in essentially the same way in both VDM and Z.

Dougal: In VDM, the whole specification gets put into a *module*: this is a structuring mechanism that makes it possible to build one module on top of others. One possible module syntax is illustrated in Appendix A. But I have to confess that this is still a subject of debate. In fact [FJ90] was written precisely because this debate is not yet settled.

Zebedee: Z also needs a modularisation mechanism and one proposal is developed in [HW93].

Florence: Does the issue of pre-conditions have any connection with the fact that I suspect the two notations handle partial functions differently?

Dougal: Yes, there is a loose connection and there are differences between Z and VDM here. In fact, I'll be interested to hear what Zebedee has to say on this point.

Let me set the scene.[7] Both operators like hd and recursively defined functions can be partial in that a simple type restriction does not indicate whether they will evaluate to a defined result for all arguments of the argument type: $\mathsf{hd}\,[\,]$ or factorial applied to minus one are examples of non-denoting terms. VDM uses non-standard logical operators which cope with possibly undefined operands: so $\mathsf{true} \lor \mathit{undefined}$ is true as is $\mathit{undefined} \lor \mathsf{true}$.

Zebedee: In [Spi88] Mike Spivey uses existential equality which always delivers a truth value – where either of the operands are undefined, the whole expression is false. This enables him to stay with classical (two-valued) logic.

Dougal: Yes, I should have said that there are a couple of different approaches where one tries to trap the undefined before it becomes an operand of a logical operator. Thus, in $\mathsf{hd}\,[\,] =_{\exists} 5$ it is possible to define the result as false. Unfortunately, the task does not end here – any relational operators need similar special versions. Moreover, the user has to keep the distinction between the logical equality and the computational equality operator of the programming language in mind. As they say 'There ain't no such thing as a free lunch'.

[7] A fuller discussion can be found in [BCJ84, CJ91, JM93].

Zebedee: Yes, but I did say Spivey took that approach; in the beginning, I believe that Jean-Raymond Abrial wanted to formalize the view that while 'the law of the excluded middle' held, for undefined formulae, one never knew which disjunct was true. (The work on the new Z standard is still evolving.)

Florence: This is all a bit technical – does it matter?

Zebedee: Not much – but it is an interesting difference!

Florence: What about recursively defined structures such as trees?

Zebedee: Before we start into the details of the definition of recursive structures, one approach often taken in Z specifications is to avoid recursive structures and use a flattened representation instead. For example, the specification of the Unix filing system [MS84] represents the hierarchical directory structure by a mapping from full path names to the file's contents. All prefixes of any path name in the domain of the map must also be in the domain of the map.

A representation of the filing system state[8] as a recursive structure can be defined by considering a directory to be a mapping from a single segment of a path name to a file, where a file is either a sequence of bytes or is itself a directory.

It is interesting to compare specifications of filing system operations on both representations. On the flat representation finding a file is simply application of the map representing the filing system state to the file name, whereas a recursive function needs to be provided for the recursive representation. If one considers updating operations, the flat representation provides even simpler descriptions of operations.

Dougal: The recursive approach description is used in [Jon90] and I'm not sure that I'd concede the advantages that Zebedee claims for the flat specification. But that's a modelling issue not a difference between the two specification languages.

Florence: So when do we need to use recursive structures?

Dougal: A good example is the specification of the abstract syntax of a programming language.

Florence: Are there examples outside the rather special field of programming languages?

Zebedee: Yes, consider a simple binary tree. This can be specified in Z via the following

$$T ::= nil \mid binnode\langle\!\langle T \times \mathbb{N} \times T \rangle\!\rangle$$

This introduces a new type T, a constant of that type nil and an (injective) constructor function *binnode*, that when given a (left sub-) tree, a natural number and a (right sub-) tree returns a tree.

Dougal: In VDM that would be

$$\begin{array}{lll} binnode :: & l & : T \\ & v & : \mathbb{N} \\ & r & : T \end{array}$$

$$T = [binnode]$$

Zebedee: There are some technical differences in the approach taken here. In Z, T is a new type, whereas for the VDM *binnode* is a new type but T is not a new type. Also, in neither

[8] We avoid consideration of inodes and links here.

Spivey [Spi92] nor the Z Base Standard are mutually recursive structures (as used in the VDM version above) allowed.

Dougal: Wouldn't that cause problems for specifying the abstract syntax of a programming language? For an Algol-like language it is common to distinguish the syntactic categories of commands and blocks, but a command may be a block and a block may contain commands.

Zebedee: If you wanted to follow the draft Z standard then you would have to merge commands and blocks into a single syntactic category and then add consistency constraints to the language to ensure that they are correctly used.

Dougal: So, it is possible to specify it, but it isn't the most natural specification.

Zebedee: True. I have to admit that I would be very tempted to extend Z to allow mutually recursive definitions.[9]

Florence: I suppose this is all linked to the semantics of the specification languages themselves.

Zebedee: To the average user, the semantics of the meta-language might not be bedtime reading. The aim has always been to base the semantics of Z on set theory; Mike Spivey gives a semantics in [Spi88] but a new semantics is being developed for the language standard.

Dougal: VDM has its origins in language description and it has to cope with reflexive domains etc. The semantics in the 'Committee Draft' of the VDM-SL standard is certainly not 'bedtime reading' but for simple operations a relatively simple set theoretic semantics would suffice.

Florence: Could you each tell me a bit about the history of your chosen specification languages?

Dougal: VDM was developed at the IBM Laboratory in Vienna. The Laboratory came into existence in 1961 when Professor Heinz Zemanek of the Technical University in Vienna decided to move his whole group to an industrial home. They had previously developed a computer called Mailüfterl at the Technical University. From 1958 the group had been increasingly involved in software projects including the construction of one of the early compilers for the ALGOL 60 programming language. As time went on they found it difficult to get adequate support for their projects and eventually joined IBM. Still in the first half of the 1960s, IBM decided to develop a new programming language for which the ambition was to replace both FORTRAN and COBOL. The language, which was at first called New Programming Language (until the National Physical Laboratories in the UK objected to the acronym – the language became known as PL/I), was clearly going to be large and it was decided that it would be useful to try to apply some formal techniques to its description.

Based on their own work – and influenced by research work by Cal Elgot, Peter Landin and John McCarthy – the Vienna group developed an operational semantics definition of PL/I which they called ULD-3 (Universal Language Description; ULD-2 was the name which had been applied to the IBM Hursley contribution to this effort; the language itself was being developed mainly from Hursley along with the early compilers. ULD-1 was a term applied to the natural language description of the language.)[10] The description of PL/I in ULD-3 style ran through three versions. These are very large documents. Operational semantics is now seen as unnecessarily complicated when compared to denotational semantics. However, to make the principles of denotational semantics applicable to a language like PL/I with arbitrary transfer

[9]The problem here is the scope of the definitions *vis a vis* constraints on the set defined by the recursive structure.

[10]VDL stands for Vienna Description Language and was a term coined by JAN Lee for the notation used in ULD-3.

of control, procedures as arguments, complicated tasking, etc. required major theoretical breakthroughs and a considerable mathematical apparatus not available at the time. The effort of the formal definition uncovered many language problems early and had a substantial influence on the shape of the language.

Towards the end of the 1960s serious attempts were made to use the ULD-3 description as the basis of compiler designs. Many problems were uncovered. The over-detailed mechanistic features of an operational semantics definition considerably complicated the task of proving that compiling algorithms were correct. But again one should be clear that an achievement was made; a series of papers was published which did describe how various programming language concepts could be mapped into implementation strategies which could be proved correct from the description. A series of proposals were made which could simplify the task of developing compilers from a semantic description. One of these was an early form of an exit construct which actually led to an interesting difference between the Vienna denotational semantics and that used in Oxford. Another important idea which arose at this time was Peter Lucas' *twin machine* proof and subsequently the observation that the ghost variable type treatment in the twin machine could be replaced by retrieve functions as a simpler way of proving that this sort of data development was correct. It is worth noting that Lucas' *twin machine* idea has been re-invented several times since: the generalization of retrieve functions to relations can be seen as equivalent to twin machines with invariants.

The person who initiated the move that pushed the Vienna group in the direction of denotational semantics was Hans Bekič's; he spent some time in England with Peter Landin at Queen Mary College.

During the period from 1971 to 1973, the Vienna group was diverted into other activities not really related to formal description. Cliff Jones at this time went back to the Hursley Laboratory and worked on a *functional* language description and other aspects of what has become known as VDM. In particular he published a development of Earley's recogniser which is one of the first reports to use an idea of data refinement. In late 1972 and throughout '73 and '74 the Vienna group (Cliff Jones returned and Dines Bjørner was recruited) had the opportunity to work on a PL/I compiler for what was then a very novel machine. They of course decided to base their development for the compiler on a formal description of the programming language. PL/I at that time was undergoing ECMA/ANSI standardisation. The Vienna group chose to write a denotational semantics for PL/I. This is the origin of the VDM work. VDM stands for Vienna Development Method. When they decided not to go ahead with the machine, IBM decided to divert the Vienna Laboratory to other activities in 1976. This led to a diaspora of the serious scientists. Dines Bjørner went to Copenhagen University then on to the Technical University of Denmark. Peter Lucas left to join IBM Research in the States. Wolfgang Henhapl left to take up a chair in Germany and Cliff Jones left to move to IBM's European System Research Institute (ESRI) in Brussels.

Cliff Jones and Dines Bjørner took upon themselves the task of making sure that something other than technical reports existed to describe the work that had gone on on the language aspects of VDM. LNCS 61 ([BJ78]) is a description of that work. At ESRI, Cliff Jones also developed the work on those aspects of VDM not specifically related to compiler development and the first book on what is now generally thought of as VDM is [Jon80]. Both of these books have now been supplanted. The language description work is best accessed in [BJ82] and the non-language work is best seen in – second edition – [Jon90].

Dines Bjørner's group at the Technical University of Denmark strenuously pursued the use of VDM for language description and he and his colleagues were responsible for descriptions of the CHILL programming language and a major effort to document the semantics of the Ada programming language. Cliff Jones spent 1979 to 81 at Oxford University (collecting a somewhat belated doctorate). This was an interesting period because Cliff and Jean-Raymond Abrial arrived within a few days of each other in Oxford and had some interesting interchanges about the evolving description technique which through many generations has been known as Z.

The non-language aspects of VDM were taken up by the STL laboratory in Harlow and, partly because of their industrial push, BSI were persuaded to establish a standardisation activity. This activity has not been easy to get going because of the differences between the pressures of the language description aspects of VDM and those who are only interested in pre/post-conditions, data reification and operation decomposition. It is to the credit of the standards committee that they have managed to bear in mind the requirements of both sorts of user and come up with a standard which embraces such a wide scope of technical ideas. There are now many books on VDM and more papers than even Dines Bjørner's energy could keep in a bibliography although Peter Gorm-Larsen has made an attempt to continue the work of keeping the key references in a single bibliography [Lar93].

The ideas in VDM have influenced several other specification languages including RAISE, COLD-K and VVSL.

Florence: Can you tell me how Z started?

Zebedee: Well, although there was a Z notation before 1979, many people associate the early development of Z with the period spent by Jean-Raymond Abrial in Oxford from 1979 to 1981. Abrial had used a paper he had written with Steve Schuman and Bertrand Meyer as lecture notes for a course given in Belfast. He was invited by Tony Hoare to Oxford, and presented similar material to the Programming Research Group (PRG) where it generated considerable interest and resulting activity. The notation described in this paper includes a basis in set theory and predicate calculus, although at this time the schema notation had not been fully developed.

Jean-Raymond Abrial was in Oxford at the same time as Cliff Jones, who had already worked on the Vienna Definition Language and the Vienna Development Method. The intention was that the two should exchange ideas and objectives and there were productive communications between the two, although in the end each pursued a distinctive path.

Florence: Is there such a thing as a Z method?

Zebedee: Z is a *notation*, and there is no official method attached to it, though there are conventions and practices that make it specially suitable for specifications written in the model-oriented style. The status of Z as a mathematical notation (rather than a method) is deliberate, and gives it flexibility and open-endedness.

Florence: How did the notation develop after the first proposals?

Zebedee: As with much of the PRG research, early development of Z centred on industrial case studies. An important early case study was CAVIAR, a visitor information system for an industrial company based on requirements from STL; other case studies carried out in the early stages included those based on the UNIX File System, the ICL Data Dictionary, and several on topics in Distributed Computing. PRG members carrying out case studies included Carroll Morgan, Ian Hayes, Bernard Sufrin, Ib Sørensen, and others. Ian Hayes, in addition to his contributions to the IBM CICS project, later collected these case studies and published them in the first book on Z [Hay93].

One of the most extensive case studies has been the use of Z for defining CICS, a transaction processing system developed by IBM. The collaboration between the PRG and the Hursley development laboratory, starting in 1982 and still continuing, has been a valuable source of information and experience for both groups.

During this early period the design of the most distinctive feature of Z, the *schema*, together with related schema operations, emerged in its present form. The Z schema notation was originally introduced as a technique for structuring large specifications and was seen as a means of naming and copying segments of mathematical text, much like a textual macro language. It was later apparent that schemas could be used more generally to define the combination of specifications, and the basic operations of schema inclusion and conjunction were extended to form the more comprehensive operations that make up what has been called the *schema calculus*.

Florence: You've talked about the PRG contribution to application case studies – what about the underlying theory?

Zebedee: In early stages of Z development, the notation was described in documents produced in the PRG and locally distributed. The complete language description, *The Z Handbook* by B. A. Sufrin [Suf88], was given only a limited circulation, and in fact the first account of the notation published in book form was in the 1987 edition of the collection of *Case Studies* (second edition [Hay93]) mentioned above. Theoretical work on the foundations of Z continued in the PRG and elsewhere, and an important contribution was provided by the D.Phil. thesis of Mike Spivey, subsequently published as a book [Spi88].

With a growing number of industrial users of Z, requests for standardisation were made at Z User Meetings in 1987 and 1988. Work was started in the Programming Research Group to establish an agreed definition of the language. Starting with the best available documentation, including [Suf88], the document produced in 1989 as a result of this work, the *Reference Manual* by J. M. Spivey became a widely accepted description of Z and provided the main starting point for the standards work described below – it is now in its second edition [Spi92].

Florence: I believe there is now a draft standard for Z – what is the status of this?

Zebedee: Towards the end of 1989 a project to develop Standards, Methods and Tools for Z was set up, with supporting funding from the UK Department of Trade and Industry. The formation of the ZIP project marked the beginning of a further stage of development, providing a stable basis for the development of national and international standards for Z. As with other projects of this kind, members of the project included both industrial and academic partners. The project was divided into four main working groups dealing with Standards, Methods and Tools – there was also a Foundations group providing theoretical support, mainly for the standards work.

The Z Standard Group developed new material for the standard, not only providing a newly written document in the style needed for a standard, but also introducing new material for the semantics (see for example [GLW91]) and logic [WB92] defined in the standard. The first draft, Version 1.0 (reference) was presented at the Z Users Meeting in December 1992 and the standards committee is now at work, reviewing and revising the document as it becomes ready for standardisation in ISO.

Meanwhile, industry users are busy using Z on projects, writing tools for Z and considering how it can be combined with other notations and methods. A good idea of the breadth and variety of interest can be gained from the Z Bibliography [Bow92].

The standards committees for Z and VDM-SL keep in touch by exchange of documents and by the appointment of liaison members. They are both subcommittees of the same BSI standards committee.

Florence: Could you give me some useful references?

Zebedee: For Z, the standard reference for the language (until the language standard appears) is Mike Spivey's [Spi92]. However, this is a language reference manual and there are some more introductory texts such as [PST91, Wor92] and a book of case studies [Hay93].

Dougal: For the non-compiler aspects of VDM, the standard reference has been [Jon90] and a case studies book is [JS90]; but [WH93] refers to Jones' book as 'austere' and either of [AI91, LBC90] might be more approachable. A good overview of VDM-SL is contained in [Daw91]; although there are several books on the language description and compiler development aspects of VDM, they haven't really come up very much in our discussion.

Florence: You have both ignored details of concrete syntax of the mathematical notation: these differences confuse some people.

Zebedee: Yes, but they are just an accident of history.

Dougal: A list of the syntactic differences has been given in a note [ISO91] from the Japanese ISO representatives.

Florence: Well, it's time for bed.

Zebedee: Boing!

Dougal: *Chases his tail for a bit before running off to bed.*

Acknowledgements

We would like to acknowledge the input from John Fitzgerald, and for permission to reuse the NDB material from [FJ90]. Peter Gorm Larsen, Lynn Marshall, Anthony Hall, Tony Hoare and Tim Clement gave us useful comments on drafts of this paper. Cliff Jones thanks the SERC for the financial support of his Senior Research Fellowship, and Ian Hayes both the financial support from the Special Studies Program from the University of Queensland and the hospitality of the Department of Computer Science at the University of Manchester, where he visited for the first half of 1993. We would also like to thank the BCS journal *Formal Aspects of Computing* for permission to reuse the NDB material from [Hay92].

References

[AI91] D. Andrews and D. Ince. *Practical Formal Methods with VDM.* McGraw-Hill, 1991.

[BCJ84] H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* Prentice Hall International, 1982.

[Bow92] J.P. Bowen. Select Z bibliography. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 367–397. Springer-Verlag, 1992.

[CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[Daw91] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[FJ90] J.S. Fitzgerald and C. B. Jones. Modularizing the formal description of a database system. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 189–210. Springer-Verlag, 1990.

[GLW91] P.H.B. Gardiner, P.J. Lupton, and Jim C.P. Woodcock. A simpler semantics for Z. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 3–11. Springer-Verlag, 1991.

[Hay92] I. J. Hayes. VDM and Z: A comparative case study. *Formal Aspects of Computing*, 4(1):76–99, 1992.

[Hay93] Ian Hayes, editor. *Specification Case Studies*. Prentice Hall International, second edition, 1993.

[Hoa91] C. A. R. Hoare. Preface. In *[PT91]*, pages vii–x, 1991.

[HW93] I. J. Hayes and L. P. Wildman. Towards libraries for Z. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop: Proceedings of the Seventh Annual Z User Meeting, London, December 1992*, Workshops in Computing. Springer-Verlag, 1993.

[ISO91] ISO. Japan's input on the VDM-SL standardization, April 1991. ISO/IEC JTC1/SC22/WG19-VDM-SL.

[JM93] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. Logic Group Preprint Series 89, Utrecht University, Department of Philosophy, April 1993.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[JS90] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[Lar93] Peter Gorm Larsen. VDM as a mature formal method. Technical report, Institute of Applied Computer Science, April 1993.

[LBC90] J. T. Latham, V. J. Bush, and I. D. Cottam. *The Programming Process: An Introduction Using VDM and Pascal*. Addison-Wesley, 1990.

[MS84] C.C. Morgan and B.A. Sufrin. Specification of the UNIX file system. *IEEE Trans. on Software Engineering*, SE-10(2): 128–142, March 1984.

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall International, 1991.

[PT91] S. Prehn and W. J. Toetenel, editors. *VDM'91 – Formal Software Development Methods. Proceedings of the 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 1991, Vol.2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Spi88] J.M. Spivey. *Understanding Z—A Specification Language and its Formal Semantics.* Cambridge Tracts in Computer Science 3. Cambridge University Press, 1988.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, second edition, 1992.

[Suf88] B. A. Sufrin. Notes for a Z handbook: Part 1 – the mathematical language, 1988. Programming Research Group, Oxford University.

[Wal90] A. Walshe. NDB. In *[JS90]*, chapter 2, pages 11–46. Prentice Hall International, 1990.

[War93] N. Ward. Adding specification constructors to the refinement calculus. In J.C.P. Woodcock and P.G. Larsen, editors, *Proceedings, Formal Methods Europe'93*, volume 670 of *Lecture Notes in Computer Science*, pages 652–670. Springer Verlag, 1993.

[WB92] J.C.P. Woodcock and S.M. Brien. W: A logic for Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 77–96. Springer-Verlag, 1992.

[Wel82] A. Welsh. The specification, design and implementation of NDB. Master's thesis, University of Manchester, October 1982.

[WH93] M. Woodman and B. Heal. *Introduction to VDM.* McGraw-Hill, 1993.

[Wor92] J. B. Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.

A VDM specification

This specification has been adapted from the NDB specification in [FJ90]. In some minor respects (e.g. optional relation names), it is more restrictive than the original [Wel82] (to which the reader is referred for a description of the operations – such as $ADDTUP$ – which are not discussed above).

module NDB
parameters
  types $Value, Esetnm, Rnm$: $Triv$
exports
  operations $ADDES, ADDENT, ADDREL, ADDTUP,$
    $DELES, DELENT, DELREL, DELTUP$
definitions
defined types

$Eid = \mathsf{token}$

$Maptp = \{\text{OneOne}, \text{OneMany}, \text{ManyOne}, \text{ManyMany}\}$

$$\begin{array}{lll} Tuple :: fv & : & Eid \\ tv & : & Eid \end{array}$$

$Relation = Tuple\text{-}\mathsf{set}$

$$\begin{array}{lll} Rinf :: tp & : & Maptp \\ r & : & Relation \end{array}$$

inv $(mk\text{-}Rinf(tp, r)) \triangleq arity\text{-}match(tp, r)$

$$\begin{array}{lll} Rkey :: nm & : & Rnm \\ fs & : & Esetnm \\ ts & : & Esetnm \end{array}$$

state

$$\begin{array}{lll} Ndb :: esm & : & Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set} \\ em & : & Eid \xrightarrow{m} Value \\ rm & : & Rkey \xrightarrow{m} Rinf \end{array}$$

inv $(mk\text{-}Ndb(esm, em, rm)) \triangleq$
  $\mathsf{dom}\, em = \bigcup \mathsf{rng}\, esm \wedge$
  $\forall rk \in \mathsf{dom}\, rm \cdot$
    $\{rk.fs, rk.ts\} \subseteq \mathsf{dom}\, esm \wedge$
    $\forall mk\text{-}Tuple(fv, tv) \in rm(rk).r \cdot fv \in esm(fs) \wedge tv \in esm(ts)$

init $(ndb) \quad \triangleq \quad ndb = mk\text{-}Ndb(\{\,\}, \{\,\}, \{\,\})$

defined functions

$arity\text{-}match(tp, r) \quad \triangleq$

$$(tp = \text{OneMany} \;\Rightarrow\; \forall t_1, t_2 \in r \cdot t_1.tv = t_2.tv \;\Rightarrow\; t_1.fv = t_2.fv) \wedge$$
$$(tp = \text{ManyOne} \;\Rightarrow\; \forall t_1, t_2 \in r \cdot t_1.fv = t_2.fv \;\Rightarrow\; t_1.tv = t_2.tv) \wedge$$
$$(tp = \text{OneOne} \;\Rightarrow\; \forall t_1, t_2 \in r \cdot t_1.fv = t_2.fv \;\Leftrightarrow\; t_1.tv = t_2.tv)$$

defined operations

$ADDES\ (es: Esetnm)$

ext wr esm : $Esetnm \xrightarrow{m} Eid$-set

pre $es \notin$ dom esm

post $esm = \overleftarrow{esm} \cup \{es \mapsto \{\,\}\}$

$DELES\ (es: Esetnm)$

ext wr esm : $Esetnm \xrightarrow{m} Eid$-set
rd rm : $Rkey \xrightarrow{m} Rinf$

pre $es \in$ dom $esm \wedge esm(es) = \{\,\} \wedge$
$\forall rk \in$ dom $rm \cdot es \neq rk.fs \wedge es \neq rk.ts$

post $esm = \{es\} \triangleleft \overleftarrow{esm}$

$ADDENT\ (memb: Esetnm\text{-set}, val: Value)\ eid: Eid$

ext wr esm : $Esetnm \xrightarrow{m} Eid$-set
wr em : $Eid \xrightarrow{m} Value$

pre $memb \subseteq$ dom esm

post $eid \notin$ dom $\overleftarrow{em} \wedge$
$em = \overleftarrow{em} \cup \{eid \mapsto val\} \wedge$
$esm = \overleftarrow{esm} \dagger \{es \mapsto \overleftarrow{esm}(es) \cup \{eid\} \mid es \in memb\}$

$DELENT\ (eid: Eid)$

ext wr esm : $Esetnm \xrightarrow{m} Eid$-set
wr em : $Eid \xrightarrow{m} Value$
rd rm : $Rkey \xrightarrow{m} Rinf$

pre $eid \in$ dom $em \wedge$
$\forall t \in \bigcup\{ri.r \mid ri \in \text{rng}\ rm\} \cdot t.fv \neq eid \wedge t.tv \neq eid$

post $esm = \{es \mapsto \overleftarrow{esm}(es) - \{eid\} \mid es \in \text{dom}\ \overleftarrow{esm}\} \wedge$
$em = \{eid\} \triangleleft \overleftarrow{em}$

$ADDREL\ (rk\colon Rkey, tp\colon Maptp)$

$\mathsf{ext\ rd}\ esm\ :\ Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}$
$\quad \mathsf{wr}\ rm\ :\ Rkey \xrightarrow{m} Rinf$

$\mathsf{pre}\ \{rk.fs, rk.ts\} \subseteq \mathsf{dom}\ esm \wedge$
$\quad rk \notin \mathsf{dom}\ rm$

$\mathsf{post}\ rm = \overleftarrow{rm} \cup \{rk \mapsto mk\text{-}Rinf(tp, \{\,\})\}$

$DELREL\ (rk\colon Rkey)$

$\mathsf{ext\ wr}\ rm\ :\ Rkey \xrightarrow{m} Rinf$

$\mathsf{pre}\ rk \in \mathsf{dom}\ rm \wedge r(rm(rk)) = \{\,\}$

$\mathsf{post}\ rm = \{rk\} \mathbin{\lhd\!\!\!-} \overleftarrow{rm}$

$ADDTUP\ (fval, tval\colon Eid, rk\colon Rkey)$

$\mathsf{ext\ wr}\ rm\ :\ Rkey \xrightarrow{m} Rinf$
$\quad \mathsf{rd}\ esm\ :\ Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}$

$\mathsf{pre}\ rk \in \mathsf{dom}\ rm \wedge$
$\quad \mathsf{let}\ mk\text{-}Rkey(nm, fs, ts) = rk\ \mathsf{in}$
$\quad \mathsf{let}\ mk\text{-}Rinf(tp, r) = rm(rk)\ \mathsf{in}$
$\quad fval \in esm(fs) \wedge tval \in esm(ts) \wedge arity\text{-}match(tp, r \cup mk\text{-}Tuple(fval, tval))$

$\mathsf{post}\ rm = \overleftarrow{rm} \dagger \{rk \mapsto \mu(\overleftarrow{rm}(rk), r \mapsto r(\overleftarrow{rm}(rk)) \cup \{mk\text{-}Tuple(fval, tval)\})\}$

$DELTUP\ (fval, tval\colon Eid, rk\colon Rkey)$

$\mathsf{ext\ wr}\ rm\ :\ Rkey \xrightarrow{m} Rinf$

$\mathsf{pre}\ rk \in \mathsf{dom}\ rm$

$\mathsf{post\ let}\ ri = \mu(\overleftarrow{rm}(rk), r \mapsto r(\overleftarrow{rm}(rk)) - \{mk\text{-}Tuple(fval, tval)\})\ \mathsf{in}$
$\qquad rm = \overleftarrow{rm} \dagger \{rk \mapsto ri\}$

$\mathsf{endmodule}\ NDB$

B Z specification

This specification has been adapted from the specification of NDB given in [Hay92].

B.1 Entities and entity sets (or types)

$[Eid, Esetnm, Value]$

Entities

$esm\colon Esetnm \mathrel{+\!\!\!\!\!\twoheadrightarrow} (\mathbb{F}\ Eid)$
$em\colon Eid \mathrel{+\!\!\!\!\!\twoheadrightarrow} Value$

$\mathrm{dom}\ em = \bigcup(\mathrm{ran}\ esm)$

$$\Delta Entities \mathrel{\widehat=} Entities \land Entities'$$

$$\Xi Entities \mathrel{\widehat=} [\Delta Entities \mid \theta Entities' = \theta Entities]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{ADDES0} \\ \hline
\Delta Entities \\
es?: Esetnm \\ \hline
es? \notin \operatorname{dom} esm \land \\
esm' = esm \cup \{es? \mapsto \{\}\} \land \\
em' = em \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{DELES0} \\ \hline
\Delta Entities \\
es?: Esetnm \\ \hline
es? \in \operatorname{dom} esm \land esm(es?) = \{\} \land \\
esm' = \{es?\} \ntriangleleft esm \land \\
em' = em \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{ADDENT0} \\ \hline
\Delta Entities \\
memb?: \mathbb{F}\ Esetnm \\
val?: Value \\
eid!: Eid \\ \hline
memb? \subseteq \operatorname{dom} esm \land \\
eid! \notin \operatorname{dom} em \land \\
em' = em \cup \{eid! \mapsto val?\} \land \\
esm' = esm \oplus \{es: memb? \bullet es \mapsto esm(es) \cup \{eid!\}\} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{DELENT0} \\ \hline
\Delta Entities \\
eid?: Eid \\ \hline
eid? \in \operatorname{dom} em \land \\
em' = \{eid?\} \ntriangleleft em \land \\
esm' = \{es: \operatorname{dom} esm \bullet es \mapsto esm(es) \setminus \{eid?\}\} \\ \hline
\end{array}$$

B.2 A single relation

$$Tuple == Eid \times Eid$$
$$Relation == Eid \leftrightarrow Eid$$

$$Maptp ::= OneOne \mid OneMany \mid ManyOne \mid ManyMany$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Rinf}} \\
\hline
tp : Maptp \\
r : Relation \\
\hline
(tp = OneOne \Rightarrow r \in Eid \rightarrowtail\!\!\!\!\!\!+\; Eid) \land \\
(tp = ManyOne \Rightarrow r \in Eid \mathrel{+\!\!\!\rightarrow} Eid) \land \\
(tp = OneMany \Rightarrow r^{\sim} \in Eid \mathrel{+\!\!\!\rightarrow} Eid) \\
\hline
\end{array}
$$

$\Delta Rinf \mathrel{\widehat=} [Rinf; Rinf' \mid tp' = tp]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{ADDTUPLE0} \\
\hline
\Delta Rinf \\
t? : Tuple \\
\hline
r' = r \cup \{t?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{DELTUPLE0} \\
\hline
\Delta Rinf \\
t? : Tuple \\
\hline
r' = r \setminus \{t?\} \\
\hline
\end{array}
$$

B.3 Multiple relations

$[Rnm]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{Rkey} \\
\hline
nm : Rnm \\
fs, ts : Esetnm \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{Ndb} \\
\hline
Entities \\
rm : Rkey \mathrel{+\!\!\!\!+\!\!\!\rightarrow} Rinf \\
\hline
\forall rk : \operatorname{dom} rm \bullet \\
\quad \{rk.fs, rk.ts\} \subseteq \operatorname{dom} esm \land \\
\quad (\forall t : (rm\ rk).r \bullet \\
\qquad first\ t \in esm(rk.fs) \land second\ t \in esm(rk.ts)) \\
\hline
\end{array}
$$

$\Delta Ndb \mathrel{\widehat=} Ndb \land Ndb'$

$\Delta REL \mathrel{\widehat=} \Delta Ndb \land \Xi Entities$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{ADDREL}} \\
\Delta REL \\
tp?: Maptp \\
rk?: Rkey \\
\hline
rk? \notin \mathrm{dom}\, rm \land \\
\{rk?.fs, rk?.ts\} \subseteq \mathrm{dom}\, esm \land \\
rm' = rm \cup \{rk? \mapsto (\mu\, Rinf \mid r = \{\} \land tp = tp?)\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{DELREL}} \\
\Delta REL \\
rk?: Rkey \\
\hline
rk? \in \mathrm{dom}\, rm \land \\
(rm\, rk?).r = \{\} \land \\
rm' = \{rk?\} \lhd rm \\
\hline
\end{array}
$$

B.4 Promotion of operations

$\Xi RM \mathrel{\widehat=} [\Delta Ndb \mid rm' = rm]$

$ADDES \mathrel{\widehat=} ADDES0 \land \Xi RM$
$DELES \mathrel{\widehat=} DELES0 \land \Xi RM$
$ADDENT \mathrel{\widehat=} ADDENT0 \land \Xi RM$
$DELENT \mathrel{\widehat=} DELENT0 \land \Xi RM$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{Promote}} \\
\Delta REL \\
rk?: Rkey \\
\Delta Rinf \\
\hline
rk? \in \mathrm{dom}\, rm \land \\
\theta Rinf = rm(rk?) \land \\
rm' = rm \oplus \{rk? \mapsto \theta Rinf'\} \\
\hline
\end{array}
$$

$ADDTUPLE \mathrel{\widehat=} (\exists\, \Delta Rinf \bullet ADDTUPLE0 \land Promote)$
$DELTUPLE \mathrel{\widehat=} (\exists\, \Delta Rinf \bullet DELTUPLE0 \land Promote)$