Under consideration for publication in Formal Aspects of Computing

VDM and Z: A Comparative Case Study

Ian Hayes

Department of Computer Science, University of Queensland, Brisbane, 4072 Australia

Keywords: VDM; Z; Specification languages; Modularisation

Abstract. The specification notations of VDM and Z are closely related. They both use model-based specification techniques and share a large part of their mathematical notation. However, the approaches taken to writing specifications differ in other, more subtle, ways.

We present a comparative case study of VDM and Z for specifying database systems. John Fitzgerald and Cliff Jones in their paper entitled "Modularising the formal description of a database system" in the proceedings of *VDM '90: VDM and Z* (LNCS Vol. 428, Springer-Verlag) provide the basis for the comparison. We present equivalent Z specifications to the VDM specifications contained in their paper.

The approach taken in writing the Z specifications is to reuse as much as possible of the Z mathematical toolkit and to build the system specification from specifications of components of the system.

In their paper, Fitzgerald and Jones emphasise their modularisation facilities. While the facilities for modularisation in Z are not as powerful, they are adequate for the specification of the database systems presented.

1. Introduction

In [FJ90], John Fitzgerald and Cliff Jones present a sequence of specifications of a binary-relation database, NDB. The first is a flat specification, the second makes use of an n-ary relation module, and the third uses an n-ary relation module with type and normalisation constraints. The main purpose of their paper is to demonstrate the use of proposed modularisation facilities for VDM. They demonstrate the reusability of their modules, they also outline specifications for an n-ary relational database with normalisation constraints (RDB), and an n-ary relation database with a two-level type hierarchy and no normalisation constraints (IS/1).

The purpose of the present paper is to provide a comparison of the VDM approach taken in [FJ90] and the approach taken in developing Z specifications for the same systems. Before going any further, we should point out that overall VDM and Z are quite similar: they are both model-based specification methods and

Correspondence and offprint requests to: Ian Hayes, Department of Computer Science, University of Queensland, Brisbane, 4072 Australia share a large part of their mathematical notation. People trained in one of these methods typically have little trouble understanding documents written using the other.

In comparing VDM and Z, there are a number of issues that we should distinguish:

- the mathematical notations,
- specifying operations,
- structuring specifications, and
- providing reusable libraries.

Section 2 begins our comparison with a Z specification of NDB. This corresponds to the flat specification of NDB in [FJ90] (which corresponds to the abstract level specification in [Wal90]). Where possible, we have used the same identifiers as [FJ90]. For this presentation, we keep the motivation and explanation of the facilities of NDB to a minimum, as our purpose is one of comparison of specifications; NDB just happens to be the vehicle for that comparison. For more detail on Z the reader is referred to either the Z reference manual [Spi89] or the collection of specification case studies [Hay87].

Section 3 gives a general model for n-ary relations in Z along with some sample operators. An outline of how a specification of NDB can be developed using n-ary relations is then presented.

Section 4 gives reasonably direct equivalents of the typing and normalisation constructs given in [FJ90], along with an outline of how these would be used to specify NDB, RDB and IS/1. Section 5 gives a simpler alternative development of the same features, but as independent primitives rather than as a bundled module. Outlines of the specifications of NDB, RDB and IS/1 are redeveloped using these primitives.

Section 6 presents a discussion of the differences in the VDM and Z approaches to specification as highlighted by the database examples.

2. NDB

The Z approach to structuring specifications is to try to build the specification from near orthogonal components. We look for ways of partitioning 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 partition the specification into three parts:

1. entities and their types or entity sets (Section 2.1),
2. a single relation (Section 2.2), and
3. multiple relations (Section 2.3).

In Section 2.4 we put these specifications together to give the final specification.

2.1. Entities and entity sets (or types)

Each entity in the database has a unique identifier taken from the set *Eid*. An entity can belong to one or more entity sets (or types). The names of these sets are taken from the set *Esetnm*. Entities have values taken from the set *Value*. These are our given sets.

$[Eid, Esetnm, Value]$

For a database we keep track of the names of entity sets and 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 data-type invariant linking them.

$$\begin{array}{|l}
\textit{Entities} \\
\hline
names : \mathbb{P}\, Esetnm \\
esm : Esetnm \leftrightarrow Eid \\
em : Eid \pfun Value \\
\hline
\operatorname{dom} esm \subseteq names \land \\
\operatorname{dom} em = \operatorname{ran} esm \\
\hline
\end{array}$$

Aside: The notation $\mathbb{P}\,\mathit{Esetnm}$ stands for the power set of *Esetnm* and is equivalent to the VDM notation *Esetnm-set*.

Rather than use a function, *esm*, from entity set names to sets of entity identifiers as in [FJ90], we have chosen to use a combination of the set of known *names* and a binary relation, *esm*, between entity set names and entity identifiers. This is more in keeping with Z style which makes use of binary relations, which are not normally available in VDM. Appendices A and B contain glossaries of Z binary relation and function notation, respectively.

The VDM version uses an optional value as the target set for *em*. This notation is not directly available in Z and, as no use is made of optional values in the specification as presented, it was thought simplest to elide the optionality. In Z, this example would normally be handled by allowing entity identifiers in the range of *esm* to be omitted from the domain of *em*. In general, optional values can be handled by having a single, additional, value in the set *Value* to stand for the *nil* case. □

We have operations to add new entity set names and to delete them. The operations are specified by giving the relationship between the 'before' state *Entities* and the 'after' state *Entities*$'$. We introduce the following delta schema for state-to-state changes.

$$\Delta \mathit{Entities} == \mathit{Entities} \land \mathit{Entities}'$$

Some later operations do not modify the entities state. We introduce the following no-change schema.

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

Aside: Here we have used the horizontal form of schema definition, with the declarations (in this case only containing included schemas) preceding the vertical bar, and the predicate part following the bar. The notation $\theta \mathit{Entities}$ stands for the value of the components of the schema *Entities* as a group. For the specification of operations the 'before' state is undecorated, the 'after' state components are primed, the names of inputs end in '?', and the names of outputs end in '!'. □

The entity set name to be added must not exist.

```
── AddES0 ──────────────────────────────
│ $\Delta \mathit{Entities}$
│ $es? : \mathit{Esetnm}$
├──────────
│ $es? \notin \mathit{names} \land$
│ $\mathit{names}' = \mathit{names} \cup \{es?\} \land$
│ $\mathit{esm}' = \mathit{esm} \land \mathit{em}' = \mathit{em}$
└──────────────────────────────────────
```

Here, a notable difference between VDM and Z arises. In Z there is no separate precondition. The VDM precondition is just included as another conjunct in the predicate describing the operation. Preconditions in Z are discussed in more detail in Section 2.5.

For the deletion of an entity set there may not be any entity of that type remaining.

```
── DelES0 ──────────────────────────────
│ $\Delta \mathit{Entities}$
│ $es? : \mathit{Esetnm}$
├──────────
│ $es? \in \mathit{names} \setminus \mathrm{dom}\, \mathit{esm} \land$
│ $\mathit{names}' = \mathit{names} \setminus \{es?\} \land$
│ $\mathit{esm}' = \mathit{esm} \land \mathit{em}' = \mathit{em}$
└──────────────────────────────────────
```

There are operations to add and delete entities. When a new entity is added its value is supplied as well as the entity set(s) that it is a member of. A new entity identifier is generated for the new entity.

```
AddEnt0
```
$$
\begin{array}{l}
\Delta Entities \\
memb? : \mathbb{P}_1\ Esetnm \\
val? : Value \\
eid! : Eid \\
\hline
memb? \subseteq names \land \\
eid! \notin \operatorname{dom} em \land \\
em' = em \cup \{eid! \mapsto val?\} \land \\
esm' = esm \cup \{tp : memb? \bullet tp \mapsto eid!\} \land \\
names' = names
\end{array}
$$

Aside: The Z notation for the set comprehension

$\{tp : memb? \bullet tp \mapsto eid!\}$,

stands for the set of pairs (maplets) with first components from the set *memb?* and second component *eid!*. In VDM this is written in the form

$\{tp \mapsto eid! \mid tp \in memb?\}$.

The Z notation for set comprehension was chosen to avoid a possible ambiguity and to be consistent with other parts of the notation. The ambiguity in the VDM form comes from whether *tp* is a new variable local to the comprehension, that ranges over the set *memb?*, or an existing variable being tested for membership in *memb?*. The Z notation uses a ':' to indicate it is a new variable, and follows the convention of placing the declaration of the variable before its use; hence the order of the declaration and the maplet are reversed in the Z version. Note that the new VDM standard avoids the above ambiguity. □

To be deleted an entity must exist.

```
DelEnt0
```
$$
\begin{array}{l}
\Delta Entities \\
eid? : Eid \\
\hline
eid? \in \operatorname{dom} em \land \\
em' = \{eid?\} \lhd em \land \\
esm' = esm \rhd \{eid?\} \land \\
names' = names
\end{array}
$$

2.2. A single relation

The relations used in NDB are binary relations between entity identifiers (rather than entity values).

$Tuple == Eid \times Eid$
$Relation == \mathbb{P}\ Tuple$

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.

$Maptp ::= one\_one \mid one\_many \mid many\_one \mid many\_many$

Aside: In Z, the set of binary relations between X and Y is written $X \leftrightarrow Y$, the set of partial functions is written $X \nrightarrow Y$, the set of one-to-one partial functions is written $X \rightarrowtail\!\!\!\!\!\!\nrightarrow Y$, and the inverse of a relation r is written $r^{\sim}$. We use these notations below. □

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

$$
\begin{array}{|l}
\textit{Rinf} \\
\hline
r : \textit{Relation} \\
tp : \textit{Maptp} \\
\hline
(tp = \textit{one\_one} \Rightarrow r \in \textit{Eid} \pinj \textit{Eid}) \land \\
(tp = \textit{many\_one} \Rightarrow r \in \textit{Eid} \pfun \textit{Eid}) \land \\
(tp = \textit{one\_many} \Rightarrow r^{\sim} \in \textit{Eid} \pfun \textit{Eid}) \land \\
(tp = \textit{many\_many} \Rightarrow r \in \textit{Eid} \leftrightarrow \textit{Eid}) \\
\hline
\end{array}
$$

Note that the last line of the predicate does not really add any extra constraint on the relation as $\textit{Eid} \leftrightarrow \textit{Eid} = \mathbb{P}(\textit{Eid} \times \textit{Eid}) = \mathbb{P}\ \textit{Tuple} = \textit{Relation}$.

The initial value of a relation is always empty and its map type is determined by an input *tp?*.

$$
\begin{array}{|l}
\textit{RinfInit} \\
\hline
\textit{Rinf} \\
tp? : \textit{Maptp} \\
\hline
r = \{\} \land tp = tp? \\
\hline
\end{array}
$$

There are operations to add tuples to, and delete tuples from a relation. These operations do not change the relation's type.

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

A tuple may be added provided it does not violate the type constraint of the relation.

$$
\begin{array}{|l}
\textit{AddTuple0} \\
\hline
\Delta \textit{Rinf} \\
t? : \textit{Tuple} \\
\hline
r' = r \cup \{t?\} \\
\hline
\end{array}
$$

Aside: In Z, the definition of *AddTuple0* includes the constraint that the before and after states both satisfy the invariant of *Rinf*. Hence there is an implicit precondition in the above definition requiring that the relation with the tuple added satisfies the type constraint specified in *Rinf*. As this constraint is non-trivial, we expand it by making the precondition of *AddTuple0* explicit. See Section 2.5 for more detail on precondition calculation in Z.

$$
\begin{array}{|l}
\textit{pre1\_AddTuple0} \\
\hline
\textit{Rinf} \\
t? : \textit{Tuple} \\
\hline
\exists\, \textit{Rinf}' \bullet tp' = tp \land r' = r \cup \{t?\} \\
\hline
\end{array}
$$

This expands to the following.

$$
\begin{array}{|l}
\textit{pre2\_AddTuple0} \\
\hline
\textit{Rinf} \\
t? : \textit{Tuple} \\
\hline
\exists\, r' : \textit{Relation} \bullet \\
\qquad r' = r \cup \{t?\} \land \\
\qquad (tp = \textit{one\_one} \Rightarrow r' \in \textit{Eid} \pinj \textit{Eid}) \land \\
\qquad (tp = \textit{many\_one} \Rightarrow r' \in \textit{Eid} \pfun \textit{Eid}) \land \\
\qquad (tp = \textit{one\_many} \Rightarrow r'^{\sim} \in \textit{Eid} \pfun \textit{Eid}) \land \\
\qquad (tp = \textit{many\_many} \Rightarrow r' \in \textit{Eid} \leftrightarrow \textit{Eid}) \\
\hline
\end{array}
$$

This schema can be simplified to remove the existential quantifier and express the precondition based solely on the inputs and initial state.

$$\begin{array}{|l}
\hline \mathit{pre3\_AddTuple0} \\
\mathit{Rinf} \\
t? : \mathit{Tuple} \\
\hline
t? \in r \lor \\
((\mathit{tp} = \mathit{one\_one} \Rightarrow \mathit{first}\ t? \notin \operatorname{dom} r \land \mathit{second}\ t? \notin \operatorname{ran} r) \land \\
\ (\mathit{tp} = \mathit{many\_one} \Rightarrow \mathit{first}\ t? \notin \operatorname{dom} r) \land \\
\ (\mathit{tp} = \mathit{one\_many} \Rightarrow \mathit{second}\ t? \notin \operatorname{ran} r) \land \\
\ (\mathit{tp} = \mathit{many\_many} \Rightarrow \mathit{true})) \\
\hline
\end{array}$$

Note that we need to allow for the cases where the tuple being added is already in the relation, as this guarantees that the resultant relation satisfies the map-type constraint. □

Any tuple may be deleted from a relation, even one not in the relation.

$$\begin{array}{|l}
\hline \mathit{DelTuple0} \\
\Delta \mathit{Rinf} \\
t? : \mathit{Tuple} \\
\hline
r' = r \setminus \{t?\} \\
\hline
\end{array}$$

2.3. Multiple relations

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

$[\mathit{Rnm}]$

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.

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

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

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

There are operations to add and delete relations. We use the following change of state abbreviations.

$$\Delta \mathit{NDB} == \mathit{NDB} \land \mathit{NDB}'$$
$$\Delta \mathit{REL} == \Delta \mathit{NDB} \land \Xi \mathit{Entities}$$

A new relation may not already exist and must relate existing entity sets. The type of the relation is given by the map-type argument (in *RinfInit*).

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddRel0}} \\
\Delta REL \\
RinfInit \\
rk? : Rkey \\
\hline
rk? \notin \operatorname{dom} rm \land \\
\{rk?.fs, rk?.ts\} \subseteq names \land \\
rm' = rm \cup \{rk? \mapsto \theta Rinf\} \\
\hline
\end{array}$$

As *Rinf* is only used to define the initial value of the new relation, we hide it in *AddRel* by existential quantification.

$AddRel == \exists\, Rinf \bullet AddRel0$

To be allowed to be deleted a relation must exist and be empty.

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

2.4. Promotion of operations

The operations given in Section 2.1 were only defined on the state *Entities*. They need to be promoted to the full *NDB* state, which includes the relations *rm*. None of the operations on entities modifies the relations, so we make use of the following promotion schema.

$\Xi RM == [\Delta NDB \mid rm' = rm]$

The promoted operations are defined as follows.

$AddES == AddES0 \land \Xi RM$
$DelES == DelES0 \land \Xi RM$
$AddEnt == AddEnt0 \land \Xi RM$
$DelEnt == DelEnt0 \land \Xi RM$

We now promote the operations on a single relation from Section 2.2 to operate on one of the relations in the database. We use the following promotion schema.

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

The promoted operations are applied to the relation named *rk?* in the database.

$AddTuple == (\exists\, \Delta Rinf \bullet AddTuple0 \land Promote)$
$DelTuple == (\exists\, \Delta Rinf \bullet DelTuple0 \land Promote)$

As *Rinf* and *Rinf′* are only used to provide access to the before and after values of the relation being updated, they are hidden in the final operations (by existentially quantifying $\Delta Rinf$).

Normally the specification of a system would include specifications of error cases as well as the normal cases given above. As these were not given in [FJ90] we have not attempted to include them. Instead we have partial operations with non-trivial preconditions.

2.5. Preconditions of operations

The *NDB* invariant constrains the entity set names of relations and the entity identifiers in relations to be known. Hence the operations *DelES* and *DelEnt* are not applicable if they would violate these constraints. In Z, the inclusion of ΔNDB in the promotion schema ΞRM automatically includes this constraint.

This is a point where the Z and VDM styles differ. The VDM style makes use of explicit preconditions and requires that constraints be included explicitly in the precondition to ensure that the invariant is maintained. In Z, these constraints are usually left implicit and the definition of the operation guarantees that the invariant is maintained.

Of course, we eventually need to make the precondition of the final operation explicit for implementation development purposes. The precondition can be *calculated* from a schema describing an operation by removing all the output and final state components from the declaration part of the schema and existentially quantifying them in the predicate part. The calculation of the precondition from the Z schema is roughly equivalent to the VDM proof obligation that the data-type invariant is maintained by the operation.

We give the calculated preconditions of the operations below. The predicate parts of these have been simplified. For *AddES* the precondition was already explicit in the predicate part.

$$
\begin{array}{|l}
\textit{pre\_AddES} \\
\hline
NDB \\
es? : Esetnm \\
\hline
es? \notin names \\
\hline
\end{array}
$$

For *DelES* the first line of the predicate was explicit in *DelES0*, while the second line in the predicate comes from the inclusion of the *NDB* invariant in the final state.

$$
\begin{array}{|l}
\textit{pre\_DelES} \\
\hline
NDB \\
es? : Esetnm \\
\hline
es? \in names \setminus \operatorname{dom} esm \land \\
(\forall rk : \operatorname{dom} rm \bullet es? \neq rk.fs \land es? \neq rk.ts) \\
\hline
\end{array}
$$

The precondition of *AddEnt* follows.

$$
\begin{array}{|l}
\textit{pre\_AddEnt} \\
\hline
NDB \\
memb? : \mathbb{P}_1\ Esetnm \\
val? : Value \\
\hline
memb? \subseteq names \land \\
\operatorname{dom} em \neq Eid \\
\hline
\end{array}
$$

The first line of the predicate was explicit in *AddEnt0*. The second line comes from the requirement that a new entity identifier *eid!* that is not in dom *em* needs to be found. This requirement is implicit in the definition of *AddEnt* above.

The precondition of *DelEnt* includes a constraint derived from the final state *NDB* invariant.

$$
\begin{array}{|l}
\textit{pre\_DelEnt} \\
\hline
NDB \\
eid? : Eid \\
\hline
eid? \in \operatorname{dom} em \land \\
(\forall rel : \operatorname{ran} rm \bullet eid? \notin \operatorname{dom} rel.r \land eid? \notin \operatorname{ran} rel.r) \\
\hline
\end{array}
$$

The preconditions of *AddRel* and *DelRel* are explicit in their definitions.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{pre\_AddRel}} \\
NDB \\
rk? : Rkey \\
tp? : Maptp \\
\hline
rk? \notin \operatorname{dom} rm \land \{rk?.fs, rk?.ts\} \subseteq names
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{pre\_DelRel}} \\
\Delta REL \\
rk? : Rkey \\
\hline
rk? \in \operatorname{dom} rm \land (rm\ rk?).r = \{\}
\end{array}$$

The precondition of *AddTuple* comes in part from the invariant on a single relation (*Rinf*) and in part from the invariant on the database (*NDB*).

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{pre\_AddTuple}} \\
NDB \\
rk? : Rkey \\
t? : Tuple \\
\hline
rk? \in \operatorname{dom} rm \land \\
(\exists\, Rinf;\ Rinf' \bullet \\
\qquad tp' = tp \land \\
\qquad \theta Rinf = rm(rk?) \land \\
\qquad r' = r \cup \{t?\})
\end{array}$$

This schema can be expanded in a manner similar to the expansion of *pre_AddTuple0* given in Section 2.2.

Deleting a tuple only requires that the named relation exist.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{pre\_DelTuple}} \\
NDB \\
rk? : Rkey \\
t? : Tuple \\
\hline
rk? \in \operatorname{dom} rm
\end{array}$$

3. N-ary Relations

Unlike [FJ90], we choose not to hide the representations of n-tuples as finite partial functions and n-ary relations as sets of tuples. The advantage of our approach is that the standard operators on functions and sets can be used directly in the specification. The disadvantage is that it becomes more difficult to change the representations of tuples and relations. But here it is worth pointing out that we are discussing specifications rather than implementations. With implementations hiding the representation is important (even essential) to allow a variety of implementations. With a specification this is not an issue. The issue is making the specification easy to understand. Reusing known operators, if they are appropriate, is one good way to reduce the complexity of a specification. In addition, a desire to change the representation at the specification level may well be accompanied by a change in the meaning of some of the exported operations on that representation, which would require careful examination of all uses of those operators in the specification.

The module facilities in [FJ90] allow a combination of,

1. grouping together of a collection of related definitions,
2. parameterisation, and
3. avoiding name clashes between identical identifiers in different modules.

In Z, the only form of modularisation currently available is to collect a group of related definitions together into a chapter in a library. A chapter may be included in a specification. As far as reasoning about the specification is concerned it is as if the chapter was textually copied into the specification. A chapter is not parameterised but the individual definitions in the chapter may have generic parameters. Each use of a generic definition is instantiated with the appropriate type parameters. Because such parameters can usually be determined from context, they do not need to be explicitly written down in the specification. Together these facilities provide a subset of the facilities of the [FJ90] modules.

The Z equivalent of the *RELATION* module in Appendix B1 of [FJ90] follows. Our treatment of n-ary relations is based on that in [SH85]. As in [FJ90] an n-tuple is represented as a finite partial function from attributes to entities:

$$NTuple[Attr, E] == Attr \pffun E.$$

As we do not hide the representation of *NTuple* we do not need to define *create* (it is the identity) and *value* (it is function application). A useful equivalence relation on n-tuples determines whether they agree on their common fields.

$$\begin{array}{l}
[Attr, E] \\
\hline
\_ \cong \_ : NTuple[Attr, E] \leftrightarrow NTuple[Attr, E] \\
\hline
\forall t1, t2 : NTuple[Attr, E] \bullet \\
\qquad t1 \cong t2 \Leftrightarrow (\operatorname{dom} t2) \lhd t1 = (\operatorname{dom} t1) \lhd t2
\end{array}$$

An n-ary relation is a set of n-tuples all of which have the same attributes:

$$\begin{array}{l}
NRelation[Attr, E] == \\
\qquad \{r : \mathbb{P}\, NTuple[Attr, E] \mid (\forall t1, t2 : r \bullet \operatorname{dom} t1 = \operatorname{dom} t2)\}
\end{array}$$

Again, as we do not hide the representation, we do not need to define *empty* (it is the empty set) and *tuples* (it is the identity).

The infix binary relation *attr_match* matches a relation with the set of attributes stored in its tuples. As in [FJ90] the empty relation matches any set of attributes.

$$\begin{array}{l}
[Attr, E] \\
\hline
\_ \ \underline{attr\_match}\ \_ : NRelation[Attr, E] \leftrightarrow (\mathbb{P}\, Attr) \\
\hline
\forall r : NRelation[Attr, E];\ as : \mathbb{P}\, Attr \bullet \\
\qquad r\ \underline{attr\_match}\ as \Leftrightarrow (\forall t : r \bullet \operatorname{dom} t = as)
\end{array}$$

Rather than provide operations to add and remove single tuples, more general union and difference operators can be introduced. In fact, for our purposes, set union and difference are adequate, and in keeping with the philosophy of reusing standard operators as much as possible we use them directly. Of course, other operators specific to n-ary relations (and not generally applicable to sets) can be defined. For example, a join operator can be defined as follows.

$$\begin{array}{l}
[Attr, E] \\
\hline
\_ \bowtie \_ : NRelation[Attr, E] \times NRelation[Attr, E] \rightarrow NRelation[Attr, E] \\
\hline
\forall r1, r2 : NRelation[Attr, E] \bullet \\
\qquad r1 \bowtie r2 = \{t1 : r1;\ t2 : r2 \mid t1 \cong t2 \bullet t1 \cup t2\}
\end{array}$$

3.1. NDB using n-ary relations

In this section we outline a revised specification of NDB using n-ary relations.
Aside: Although giving the revised specification is worthwhile for comparison purposes, I have reservations about specifying a binary-relation database, such as NDB, in terms of n-ary relations. Firstly, Z has a standard chapter for binary relations with a rich set of operators and associated laws, which it seems a pity not to make use of.

Secondly and more generally, the n-ary relation model and operators are more complex than those for binary relations. I suspect that, in a more complete specification, one would end up defining equivalents of many of the binary-relation operators available in Z in terms of the n-ary relation representation of binary relations. For example, a useful operator on binary relations is relational composition. If composition is required in the specification, and especially if it is used more than once, it will make sense to define a composition operator, because writing out a composition directly in terms of n-ary relations is complicated.

Once enough binary-relation operator equivalents are introduced, one is no longer making use of the n-ary relations model, but rather a new binary relation model (that just happens to be defined in terms of n-ary relations). □

The state and operations on entities from Section 2.1 remain unchanged, but the state and operations on relations make use of n-ary relations rather than binary relations. To represent an NDB binary relation as an n-ary relation we use the attributes names *FS* and *TS* to denote the 'from' and 'to' components:

$$Fsel ::= FS \mid TS$$

NDB tuples and relations can be defined in terms of n-tuples and n-ary relations with these attributes.

$$\begin{array}{l} NDB\_Tuple == \{t : NTuple[Fsel, Eid] \mid \operatorname{dom} t = Fsel\} \\ NDB\_Relation == \mathbb{P}\, NDB\_Tuple \end{array}$$

As a consequence of this definition we have the following equivalence.

$$NDB\_Relation = \{r : NRelation[Fsel, Eid] \mid r \ \underline{attr\_match} \ Fsel\}$$

The equivalent of *Rinf* using the new model is *NRinf*.

$$\begin{array}{|l} \text{\_\_} NRinf \text{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\ r : NDB\_Relation \\ tp : Maptp \\ \hline (tp = one\_one \Rightarrow \\ \qquad (\forall t1, t2 : r \bullet t1(FS) = t2(FS) \Leftrightarrow t1(TS) = t2(TS))) \land \\ (tp = many\_one \Rightarrow \\ \qquad (\forall t1, t2 : r \bullet t1(FS) = t2(FS) \Rightarrow t1(TS) = t2(TS))) \land \\ (tp = one\_many \Rightarrow \\ \qquad (\forall t1, t2 : r \bullet t1(TS) = t2(TS) \Rightarrow t1(FS) = t2(FS))) \land \\ (tp = many\_many \Rightarrow true) \\ \hline \end{array}$$

Aside: Note that we can no longer make use of the existing Z notation for describing functions and one-to-one functions. Instead of using the higher-level well-known (to Z users) concepts, we have to make use of predicates on the tuples of the relations that essentially redefine these concepts. □

All the schemas used earlier can be upgraded to use this new state by simply replacing all occurrences of *Rinf* with *NRinf* and *Tuple* with *NDB_Tuple*. There is one exception, the *NDB* state, which we redefine as *NNDB*.

$$\begin{array}{|l} \text{\_\_} NNDB \text{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\ Entities \\ rm : Rkey \nrightarrow NRinf \\ \hline \forall rk : \operatorname{dom} rm \bullet \\ \qquad \{rk.fs, rk.ts\} \subseteq names \land \\ \qquad (\forall t : (rm\ rk).r \bullet \\ \qquad\qquad (rk.fs, t(FS)) \in esm \land (rk.ts, t(TS)) \in esm) \\ \hline \end{array}$$

The specifications given in this section are a little more complex than those for *Rinf* and *NDB*, but from what we can see here there is really little difference. If we had to specify an operation for, say, composing two binary relations, then we would have to define such an operator on our n-ary relation representation, whereas with a binary relation representation such operators are already available.

4. Typing and Normalisation

The only differences between the modules *TYPED-RELATION* and *RELATION* in [FJ90] are the tighter constraints on tuples and relations in *TYPED-RELATION* and the additional parameters to this module necessary to specify the constraints.

The approach we take here is to leave the definitions of n-ary relations as they are, but only use them on relations that satisfy the necessary constraints. Our equivalent of *TYPED-RELATION* is the following generic schema *Typed_Relation*. It constrains the tuples of the relation to have attributes of a particular type, as well as the whole relation to a normalisation constraint. The normalisation constraint consists of a set of functional dependency constraints, each of which requires that an attribute, *f*, of the relation is functionally dependent on (uniquely determined by) a set, *s*, of other attributes.

$$\mathit{Dependency}[\mathit{Attr}] == (\mathbb{P}\,\mathit{Attr}) \times \mathit{Attr}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Typed\_Relation}[E, \mathit{Etp}, \mathit{Attr}]} \\
r : \mathit{NRelation}[\mathit{Attr}, E] \\
\mathit{tpm} : \mathit{Attr} \mathrel{+\!\!\!\twoheadrightarrow} \mathit{Etp} \\
\mathit{tpc} : E \leftrightarrow \mathit{Etp} \\
\mathit{norm} : \mathbb{P}\,\mathit{Dependency}[\mathit{Attr}] \\
\hline
(\forall m : r \bullet \mathrm{dom}\, m = \mathrm{dom}\, \mathit{tpm} \land \\
\qquad (\forall a : \mathrm{dom}\, \mathit{tpm} \bullet (m(a), \mathit{tpm}(a)) \in \mathit{tpc})) \land \\
(\forall s : \mathbb{P}\,\mathit{Attr};\ f : \mathit{Attr} \bullet (s, f) \in \mathit{norm} \Rightarrow \\
\qquad s \cup \{f\} \subseteq \mathrm{dom}\, \mathit{tpm} \land \\
\qquad (\forall t1, t2 : r \bullet s \lhd t1 = s \lhd t2 \Rightarrow t1(f) = t2(f)))
\end{array}$$

Aside: The above schema follows the structure used in [FJ90]. Upon closer examination, however, the components *tpm* and *tpc* are only used in combination, and in a way that only uses the entity type (*Etp*) indirectly. The above may be simplified by combining the two components into a single type checking relation *tpmc* which gives the entities that may be associated with an attribute. The set *Etp* is no longer needed as a parameter. The simplified schema follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Typed\_Relation}2[E, \mathit{Attr}]} \\
r : \mathit{NRelation}[\mathit{Attr}, E] \\
\mathit{tpmc} : \mathit{Attr} \leftrightarrow E \\
\mathit{norm} : \mathbb{P}\,\mathit{Dependency}[\mathit{Attr}] \\
\hline
(\forall m : r \bullet \mathrm{dom}\, m = \mathrm{dom}\, \mathit{tpmc} \land m \subseteq \mathit{tpmc}) \land \\
(\forall s : \mathbb{P}\,\mathit{Attr};\ f : \mathit{Attr} \bullet (s, f) \in \mathit{norm} \Rightarrow \\
\qquad s \cup \{f\} \subseteq \mathrm{dom}\, \mathit{tpmc} \land \\
\qquad (\forall t1, t2 : r \bullet s \lhd t1 = s \lhd t2 \Rightarrow t1(f) = t2(f)))
\end{array}$$

□

4.1. NDB using typed relations

For NDB, a typed relation is used as part of the information associated with a single relation. In addition, the ‘from’ and ‘to’ sets, and the map type are included. The component *tpm* of a typed relation is derived from the ‘from’ and ‘to’ sets, and the normalisation component of the typed relation is derived from the map type.

$$\begin{array}{|l}
\hline \textit{TRinf} \\
\textit{Typed\_Relation}[\textit{Eid}, \textit{Esetnm}, \textit{Fsel}] \\
\textit{fs}, \textit{ts} : \textit{Esetnm} \\
\textit{mtp} : \textit{Maptp} \\
\hline
\textit{tpm} = \{\textit{FS} \mapsto \textit{fs}, \textit{TS} \mapsto \textit{ts}\} \land \\
(\textit{mtp} = \textit{many\_many} \Rightarrow \textit{norm} = \{\}) \land \\
(\textit{mtp} = \textit{many\_one} \Rightarrow \textit{norm} = \{(\{\textit{FS}\}, \textit{TS})\}) \land \\
(\textit{mtp} = \textit{one\_many} \Rightarrow \textit{norm} = \{(\{\textit{TS}\}, \textit{FS})\}) \land \\
(\textit{mtp} = \textit{one\_one} \Rightarrow \textit{norm} = \{(\{\textit{TS}\}, \textit{FS}), (\{\textit{FS}\}, \textit{TS})\}) \\
\hline
\end{array}$$

Aside: In NDBRELATION in Appendix C.2 of [FJ90] an additional component (parameter) *esm* is used. Although we could include it here it seems simpler to include it in *TNDB* below, because it is the same for all relations. □

The NDB state using typed relations defines the 'from' and 'to' sets for each relation, as well as the type checking relation *tpc*, which is the inverse of *esm*.

$$\begin{array}{|l}
\hline \textit{TNDB} \\
\textit{Entities} \\
\textit{rm} : \textit{Rkey} \pfun \textit{TRinf} \\
\hline
\forall \textit{rk} : \operatorname{dom} \textit{rm} \bullet \\
\quad \{\textit{rk.fs}, \textit{rk.ts}\} \subseteq \textit{names} \land \\
\quad (\textit{rm}\ \textit{rk}).\textit{fs} = \textit{rk.fs} \land (\textit{rm}\ \textit{rk}).\textit{ts} = \textit{rk.ts} \land \\
\quad (\textit{rm}\ \textit{rk}).\textit{tpc} = \textit{esm}^{\sim} \\
\hline
\end{array}$$

These definitions are more complex than either of the previous two sets of equivalents definitions given earlier. The properties of NDB binary relations take a little more figuring out than for the representation based on binary relations.

If we compare the above with Appendix C of [FJ90] we have avoided revising the n-ary relations chapter to include type constraints. We leave the n-ary relation operators in their general form but only use them on suitably constrained relations.

4.2. RDB using typed relations

RDB requires an n-ary relation rather than a binary relation. The n-ary relation associates entities with each of a number of attributes. We assume we are given the set *Etp* of entity types, and the set *FSel* of attribute identifiers.

$[\textit{Etp}, \textit{FSel}]$

We assume that there is a fixed relationship, *tpch*, between values and entity types.

$$\begin{array}{|l} \textit{tpch} : \textit{Value} \leftrightarrow \textit{Etp} \end{array}$$

The definition of the single relation state in terms of a typed relation follows.

$$\begin{array}{|l}
\hline \textit{RRinf} \\
\textit{Typed\_Relation}[\textit{Value}, \textit{Etp}, \textit{FSel}] \\
\textit{key} : \mathbb{P}\, \textit{FSel} \\
\hline
\textit{key} \subseteq \operatorname{dom} \textit{tpm} \land \\
\textit{tpc} = \textit{tpch} \land \\
\textit{norm} = \{a : (\operatorname{dom} \textit{tpm}) \setminus \textit{key} \bullet (\textit{key}, a)\} \\
\hline
\end{array}$$

Aside: As *Typed_Relation* has a component *tpm* which is identical to the component *tp* used in *Rinf* in [FJ90], we omit *tp* in *RRinf*. We have also avoided the introduction of an equivalent to the RDBRELATION in [FJ90]. Its main purpose is to define the normalisation, but that can be done directly in *RRinf* using our approach. □

The RDB state for multiple relations is a set of relations identified by names from the set *Rnm*.

$$RDB == Rnm \pfun RRinf$$

4.3. IS/1 using typed relations

IS/1 makes use of an extra level of typing information: type names taken from the set *Tpnm*.

$$[Tpnm]$$

We make use of the fixed type relationship, *tpch*, defined above for RDB.

A single IS/1 relation is a typed relation with no normalisation constraint.

$$\begin{array}{|l}
\textit{IRinf} \\
\hline
Typed\_Relation[Value, Etp, FSel] \\
tp : FSel \ffun Tpnm \\
\hline
norm = \{\} \land tpc = tpch \\
\hline
\end{array}$$

The state for IS/1 consists of a mapping giving entity types for type names, and the set of named relations. For each relation the type names of its attributes must be known and the type constraint on each attribute is determined by the composition of the attribute's type name and the type name's entity type.

$$\begin{array}{|l}
\textit{IS1} \\
\hline
tm : Tpnm \pfun Etp \\
rm : Rnm \pfun IRinf \\
\hline
\forall rel : \mathrm{ran}\, rm \bullet \\
\qquad \mathrm{ran}\, rel.tp \subseteq \mathrm{dom}\, tm \land \\
\qquad rel.tpm = tm \circ rel.tp \\
\hline
\end{array}$$

5. Typing and normalisation revisited

The purpose of typed relations is to allow constraints to be applied to n-tuples and n-ary relations. In TYPED-RELATION this is done by providing a set of parameters to cope with the general case. There are two problems with this approach:

1. to allow for the general case the parameterisation is quite complex; this makes it harder to understand the specification, especially for the cases involving simple constraints, and
2. even though an attempt has been made to provide a general mechanism for constraining relations, there is no guarantee that it copes with the constraints required for a new application.

An alternative approach is to provide more primitive mechanisms that may be put together to provide the desired constraints. For the examples considered in this paper we require mechanisms for specifying type constraints on the tuples in a relation, and for specifying the normalisation constraints on the relation.

The type constraints on tuples for NDB, RDB and IS/1 are quite straightforward if we use a type checking relation similar to *tpmc* in *Typed_Relation2*. For our alternative specifications we use the following generic schema *Typed_Relation_x* which is similar to *Typed_Relation2* except that normalisation has been left out.

$$\begin{array}{|l}
\textit{Typed\_Relation\_x}[Attr, E] \\
\hline
r : NRelation[Attr, E] \\
type : Attr \rel E \\
\hline
\forall m : r \bullet \mathrm{dom}\, m = \mathrm{dom}\, type \land m \subseteq type \\
\hline
\end{array}$$

Aside: Although this generic schema is used in the revised versions of NDB, RDB and IS/1 below, I am not sure that it warrants the status of being included in a specification library. I suspect that, if I were developing only one of NDB, RDB or IS/1, I would not make use of *Typed_Relation_x*, but rather would include its components and constraints (or possibly more appropriate equivalents) explicitly in the corresponding *Rinf* definition. □

For normalisation it is useful to introduce some notation for specifying functional dependencies between attributes. A single functional dependency specifies that one attribute, *f*, is functionally dependent on a set of attributes, *s*. That is, any two tuples in a relation which have the same values for all the attributes in *s* have the same value for their *f* attribute as well. We introduce the infix operator *determines* and write

$$s \ \text{determines} \ f,$$

for the set of all relations in which the attribute *f* is functionally dependent on the attributes in *s*. Hence we may write

$$r \in (s \ \text{determines} \ f),$$

to indicate that relation *r* satisfies this functional dependency constraint. The definition of *determines* follows.

$$\begin{array}{l}
[Attr, E] \\
\hline
\_ \ \text{determines} \ \_ : Dependency[Attr] \rightarrow (\mathbb{P}\, NRelation[Attr, E]) \\
\hline
\forall s : \mathbb{P}\, Attr;\ f : Attr \bullet \\
\quad s \ \text{determines}\, f = \{r : NRelation[Attr, E] \mid \\
\qquad\qquad (\forall t : r \bullet s \cup \{f\} \subseteq \operatorname{dom} t) \wedge \\
\qquad\qquad (\forall t1, t2 : r \bullet s \lhd t1 = s \lhd t2 \Rightarrow t1(f) = t2(f))\}
\end{array}$$

This definition is logically part of the library chapter for n-ary relations developed in Section 3.

5.1. NDB revisited

For a single NDB relation we use our new typed relation plus normalisation constraints derived from the map type of the relation.

$$\begin{array}{l}
TRinf\_x \\
\hline
Typed\_Relation\_x[Fsel, Eid] \\
mtp : Maptp \\
\hline
(mtp = many\_many \Rightarrow true) \wedge \\
(mtp = many\_one \Rightarrow r \in (\{FS\} \ \text{determines} \ TS)) \wedge \\
(mtp = one\_many \Rightarrow r \in (\{TS\} \ \text{determines} \ FS)) \wedge \\
(mtp = one\_one \Rightarrow \\
\quad r \in (\{TS\} \ \text{determines} \ FS) \cap (\{FS\} \ \text{determines} \ TS))
\end{array}$$

The NDB state now specifies the type constraint by defining *type* for each relation. The *type* is derived from the entity sets for the 'from' and 'to' entity set names.

$$\begin{array}{l}
TNDB\_x \\
\hline
Entities \\
rm : Rkey \pfun TRinf\_x \\
\hline
\forall rk : \operatorname{dom} rm \bullet \\
\quad \{rk.fs, rk.ts\} \subseteq names \wedge \\
\quad (rm \ rk).type = \{FS \mapsto rk.fs, TS \mapsto rk.ts\} \fatsemi esm
\end{array}$$

Aside: We use (forward) relation composition ('$\fatsemi$') of two relations to specify the *type*. The first relation maps an attribute identifier (*FS* or *TS*) to the corresponding entity set name, and the relation *esm* maps the entity set name to all the corresponding entity identifiers. The resulting relation maps an attribute identifier to all the corresponding allowable entity identifiers. Note that $R1 \fatsemi R2$ is the same as $R2 \circ R1$, if we generalise '$\circ$' to composition of relations. □

5.2. RDB revisited

For RDB we assume that there is a fixed type checking relation *EntityType*, which specifies the allowable values for every entity type.

$$
\begin{array}{|l}
\mathit{EntityType} : \mathit{Etp} \leftrightarrow \mathit{Value} \\
\hline
\mathrm{dom}\,\mathit{EntityType} = \mathit{Etp}
\end{array}
$$

A single relation has a *key* on which all other attributes are functionally dependent, and each attribute of the relation has an entity type.

$$
\begin{array}{|l}
\hline
\mathit{RRinf\_x} \\
\mathit{Typed\_Relation\_x}[\mathit{FSel}, \mathit{Value}] \\
\mathit{key} : \mathbb{P}\,\mathit{FSel} \\
\mathit{tp} : \mathit{FSel} \twoheadrightarrow\!\!\!\!\!\!+\; \mathit{Etp} \\
\hline
\mathit{key} \subseteq \mathrm{dom}\,\mathit{tp} \land \\
(\forall\, a : (\mathrm{dom}\,\mathit{tp}) \setminus \mathit{key} \bullet r \in (\mathit{key}\ \mathrm{determines}\ a)) \land \\
\mathit{type} = \mathit{tp} \mathbin{⨾} \mathit{EntityType} \\
\hline
\end{array}
$$

The *type* of a relation is derived from the entity type of the attributes composed with the values of the entity type.

The RDB state is a set of named relations.

$$\mathit{RDB\_x} == \mathit{Rnm} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{RRinf\_x}$$

5.3. IS/1 revisited

For IS/1 we assume the same *EntityType* relationship as for RDB. For a single relation each attribute is associated with a named type. As there are no normalisation constraints on IS/1 relations, our specification makes no mention of normalisation or functional dependencies.

$$
\begin{array}{|l}
\hline
\mathit{IRinf\_x} \\
\mathit{Typed\_Relation\_x}[\mathit{FSel}, \mathit{Value}] \\
\mathit{tp} : \mathit{FSel} \twoheadrightarrow\!\!\!\!\!\!+\; \mathit{Tpnm} \\
\hline
\mathrm{dom}\,\mathit{type} = \mathrm{dom}\,\mathit{tp} \\
\hline
\end{array}
$$

The IS/1 state consists of a set of named relations plus a mapping from type names to entity types. Each relation may only use known type names, and each attribute's *type* is determined from the attribute's type name via the type name's entity type, and the *EntityType* relationship.

$$
\begin{array}{|l}
\hline
\mathit{IS1\_x} \\
\mathit{tm} : \mathit{Tpnm} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{Etp} \\
\mathit{rm} : \mathit{Rnm} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{IRinf\_x} \\
\hline
\forall\, \mathit{rel} : \mathrm{ran}\,\mathit{rm} \bullet \\
\qquad \mathrm{ran}\,\mathit{rel.tp} \subseteq \mathrm{dom}\,\mathit{tm} \land \\
\qquad \mathit{rel.type} = \mathit{rel.tp} \mathbin{⨾} \mathit{tm} \mathbin{⨾} \mathit{EntityType} \\
\hline
\end{array}
$$

6. Discussion

In this section we overview the differences in the VDM and Z approaches to specification as highlighted by the database examples.

In Z, the approach is to try to partition the state so that operations are only specified on that part of the state they need. In [FJ90], the approach is to give a monolithic description of the complete state before specifying any operations, although the state itself is structured. To cope with operations that only access and/or update part of the state VDM makes use of read and write imports of state components to each operation. The Z approach has the advantage of being able to build up the specification from simpler components. The specification of these components does not require the prior development of the complete state. A disadvantage is that the operations on part of the state need to be promoted to the full system state.

In Z, binary relations are a standard component of the mathematical toolkit. In VDM these are represented as either a Boolean function on a pair of arguments, or as a set of pairs. Z models a binary relation as a set of pairs, but the real difference is that the standard Z toolkit provides a comprehensive set of operators for manipulating binary relations. In fact, functions in Z are a specialisation of relations and most of the operators on functions in VDM (with one or two notable exceptions, such as function application) are defined in Z as a more general operator on binary relations.

In VDM, the precondition of each operation must be explicitly included as a separate component of the specification of an operation, whereas in Z it is implicit in the single predicate that specifies the operation. In Z, the data-type invariant is included for both the 'pre' and 'post' states of the specification of an operation. Hence the operation is specified to maintain the data-type invariant. However, achieving the data-type invariant on the post state can place a restriction on the possible pre states and hence the precondition. The precondition of a Z operation can be *calculated* from its specification. In VDM, the equivalent of this calculation is performing the proof that the operation is satisfiable.

The approach to providing a mathematical toolkit for n-ary relations is quite different. We have chosen not to hide the representation of n-ary relations and to make use of the standard operators on functions and sets, rather than the approach taken in [FJ90] where the representation of n-tuples and n-ary relations is hidden and a new set of operators are defined to work with the new data types. Introducing a new set of operators on n-ary relations has the disadvantage that it requires a reader of the specification to become familiar with the new operators before they can understand the specification, but has the advantage that the representation of n-ary relations is hidden. We would argue that hiding of data-type representations, while essential for implementations, is not essential for specifications. For implementations, it allows the module's representation to be changed (and the module to be improved) without any of the users of the module needing to know (provided the meaning of the module is not changed). For specifications, changes to the abstract representation of a data type are usually accompanied by changes in the meanings (perhaps minor, but nonetheless significant) of the operators of the module. Another advantage of not hiding the representation is that the general laws for sets and functions can be used directly in any reasoning about n-ary relations, although some new laws specific to n-ary relations may be desirable as well. If the representation is hidden then not only do the new laws specific to n-ary relations have to be established, but the general laws for sets and functions have to be established for n-ary relations. These latter laws are just repetitions of the more general laws for a restricted case. Their proofs are trivial.

The exact meaning of the parameterisation mechanism in [FJ90] is not clear, so this paragraph is based on my interpretation of its meaning. The parameterisation mechanism seems to allow both instantiation of the whole module with specific parameters, as well as use without parameters specified (and in some cases both, but I am not sure what this means). In the latter case, each operator is instantiated with the necessary parameters when it is used. In Z, there is no ability to define a module with generic parameters to the whole module; rather, each definition may have generic parameter sets. Each use of a generic definition is instantiated with the appropriate sets. In practice, the generic sets can commonly be deduced from the context, and are elided when this can be done. The generic sets of a Z definition are equivalent to the *Triv* parameter types in [FJ90]. The individual instantiations correspond closely to the use of generic parameters in [FJ90].

There is no direct equivalent in Z of the value and function parameters to modules in [FJ90], although a generic schema definition can be used to achieve the same effect in many cases. For the *Typed_Relation* in Section 4 the parameters of the [FJ90] specification are included as components of the generic schema. When the generic schema is used its generic sets are supplied as parameters, and the values of the components are defined by the predicate part of the including schema. This also makes these components directly available for other specification purposes.

To define typed relations, [FJ90] introduces a new module and redefines all the operators to work on the new, more constrained representations. In Z, we avoid redefining the n-ary relation module for typed relations, by just constraining the n-ary relations used to satisfy the type constraints. As the preconditions of operations are implicit, they are automatically strengthened where necessary so that the final state satisfies the additional constraint.

We take the approach that it is better to provide primitives that can be combined to give the final specification, rather than provide a parameterised general solution that is instantiated for each use. For example, in IS/1 there are no normalisation constraints. In [FJ90] this is specified by setting $norm = \{\}$. To understand the meaning of the specification, however, it is necessary to understand the use of normalisation in TYPED-RELATION in order to realise an empty set of normalisation constraints puts no constraint on the relations. The alternative approach in Section 5.3 avoids any mention of normalisation, as it is not required. This makes the specification easier to understand.

7. Conclusions

In practice, Z and VDM are closely related methods and many of the differences discussed above are more differences in approach than real differences in the capabilities of the notations and methods used. The approach taken in the Z specifications presented in this paper has been to reuse as much as possible of the standard Z toolkit, and to build the specification from primitives and components. This has allowed us to simplify some aspects of the specifications.

When we turn to the question of comparing the modularisation facilities of Z and [FJ90], it is clear that there are features available in [FJ90] that are not available in Z. For example, modularisation may be used to avoid name clashes and to provide one set of parameters for a group of related definitions. These desirable facilities are lacking in Z, but the database specifications covered in this paper can be adequately described without these facilities. Hence, unfortunately, these examples do not provide a good basis for discussing those aspects of [FJ90] modularisation that are not available in Z. One Z specification that cries out for these features is CAVIAR [FS87] where a specification of a resource control system is instantiated for a number of different types of resources. Perhaps CAVIAR would make a good example for further investigation of modularisation facilities.

Acknowledgements

The motivation for this paper came from a request from Cliff Jones for a comparison of the VDM and Z approaches to modularising specifications. I would like to thank John Fitzgerald, Cliff Jones, Luke Wildman and the referees of this paper for useful comments on earlier drafts of this paper. This paper was type checked using Mike Spivey's *f*UZZ program.

References

[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*, pages 189–210. Springer-Verlag, 1990. Lecture Notes in Computer Science, Vol. 428.

[FS87] L. W. Flinn and I. Holm Sørensen. CAVIAR: A case study in specification. In I. J. Hayes, editor, *Specification Case Studies*, pages 141–188. Prentice Hall International, 1987.

[Hay87] I. J. Hayes, editor. *Specification Case Studies.* Prentice Hall International, 1987.

[SH85] B. A. Sufrin and J. Hughes. A Tutorial Introduction to Relational Algebra. Technical report, Programming Research Group, Oxford University, UK, July 1985. In the Z Package.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1989.

[Wal90] A. Walshe. NDB: The formal specification and rigorous design of a single-user database system. In C. B. Jones and R. C. F. Shaw, editors, *Case Studies in Systematic Software Development*, pages 11–45. Prentice Hall International, 1990.

A. Binary Relations

A binary relation is modelled by a set of ordered pairs. Hence operators defined for sets can be used on relations. Let X, Y, and Z be sets; $x : X$; $y : Y$; S be a subset of X; T be a subset of Y; and R a relation between X and Y.

$X \leftrightarrow Y$	$== \mathbb{P}(X \times Y)$ The set of relations between X and Y.
$x \underline{R} y$	$== (x, y) \in R$ x is related by R to y.
$x \mapsto y$	$== (x, y)$
$\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$	$== \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ The relation relating x_1 to y_1, x_2 to y_2, …, and x_n to y_n.
$\mathrm{dom}\, R$	$== \{x : X \mid (\exists\, y : Y \bullet x \underline{R} y)\}$ The domain of a relation: the set of x components that are related to some y.
$\mathrm{ran}\, R$	$== \{y : Y \mid (\exists\, x : X \bullet x \underline{R} y)\}$ The range of a relation: the set of y components that some x is related to.
$R_1 \fatsemi R_2$	$== \{x : X;\ z : Z \mid (\exists\, y : Y \bullet x \underline{R_1} y \land y \underline{R_2} z)\}$ Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$. The composition relates x to z if there is some y such that x is related to y by R_1 and y is related to z by R_2.
$R_1 \circ R_2$	$== R_2 \fatsemi R_1$ Relational composition. This form is primarily used when R_1 and R_2 are functions.
$R^\sim$	$== \{y : Y;\ x : X \mid x \underline{R} y\}$ Transpose of a relation R. $R^\sim$ relates y to x if and only if R relates x to y.
$\mathrm{id}\, S$	$== \{x : S \bullet x \mapsto x\}$ Identity function on the set S.
$R(\!\vert\, S \,\vert\!)$	$== \{y : Y \mid (\exists\, x : S \bullet x \underline{R} y)\}$ Image of the set S through the relation R.
$S \lhd R$	$== \{x : X;\ y : Y \mid x \in S \land x \underline{R} y\}$ Domain restriction: the relation R with its domain restricted to the set S.
$S \ntriangleleft R$	$== (X \setminus S) \lhd R$ Domain exclusion: the relation R with the members of S excluded from its domain.
$R \rhd T$	$== \{x : X;\ y : Y \mid x \underline{R} y \land y \in T\}$ Range restriction to T.
$R \ntriangleright T$	$== R \rhd (Y \setminus T)$ Range exclusion: the relation R with the members of T excluded from its range.
$R_1 \oplus R_2$	$== (\mathrm{dom}\, R_2 \ntriangleleft R_1) \cup R_2$ Overriding; $R_1, R_2 : X \leftrightarrow Y$.

B. Functions

A function is a relation with the property that each member of its domain is associated with a unique member of its range. As functions are relations, all the operators defined above for relations also apply to functions. Let X and Y be sets, and T be a subset of X (i.e. $T : \mathbb{P}\, X$).

$f\ t$	The function f applied to t. A function f is a set of pairs with each member of its domain associated with a unique member of its range. $f\ t$ is only defined provided $t \in \mathrm{dom} f$, and its value is the unique value in the range associated with the value t in its domain: $f\ t = y \Leftrightarrow (t, y) \in f$.
$X \nrightarrow Y$	$== \{f : X \leftrightarrow Y \mid (\forall\, x : \mathrm{dom} f \bullet (\exists_1\, y : Y \bullet x \underline{f} y))\}$ The set of partial functions from X to Y.

$X \rightarrow Y$	$== \{f : X \pfun Y \mid \mathrm{dom}\, f = X\}$ The set of total functions from X to Y.
$X \pinj Y$	$== \{f : X \pfun Y \mid (\forall\, y : \mathrm{ran}\, f \bullet (\exists_1\, x : X \bullet x \underline{f}\ y))\}$ The set of partial one-to-one functions (partial injections) from X to Y.
$X \ffun Y$	$== \{f : X \pfun Y \mid f \in \mathbb{F}(X \times Y)\}$ The set of finite partial functions from X to Y.