SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-10

On transferring VDM verification techniques to Z

Peter Lindsay

March 1994

Phone: +61 7 365 1003
Fax: +61 7 365 1533

On transferring VDM verification techniques to Z

Peter A. Lindsay

Software Verification Research Centre,
University of Queensland, St Lucia, Queensland 4072, Australia
email: pal@cs.uq.edu.au

Abstract. This paper discusses some of the necessary prerequisites for transferring specification analysis and verification techniques from VDM to Z. It starts by comparing Z and VDM in terms of the mathematical and specification notations they use. It then explains the VDM approach to reasoning about specifications, as supported by the `mural` tool-set, and compares VDM's Logic of Partial Functions with Classical Logic. It outlines VDM proof obligations for checking consistency and completeness of specifications, and illustrates their use on a small example, comparing the results with a Z-like analysis. The paper concludes with a brief discussion of how the W logic for Z might be modified for LPF.

1 Introduction

1.1 Background and scope

Recent work [2] on proof in VDM, in conjunction with experience using the `mural` support environment [13], has demonstrated a practical approach to formal analysis of VDM specifications based on:

- an axiomatic semantics for a large subset of the VDM specification language VDM-SL [4], presented in Natural Deduction style;
- a Logic of Partial Functions, LPF, which emphasizes the treatment of undefinedness;
- a process for generating proof obligations to check the well-formedness and mathematical consistency of specifications (in a sense to be defined below);
- a large store of re-usable theorems.

The mathematical language in which theorem proving takes place is almost indistinguishable from the mathematical component of VDM-SL. The main advantages of this integrated approach are that:

- formal proofs are expressed in a very natural form (similar to that of rigorous proofs in mathematical textbooks) for which effective machine support can be provided;
- analysis for well-formedness reveals certain incompletenesses in specifications.

The purpose of this paper is to bring the main points of the approach to the attention of the Z community and to sketch how it might be transferred to Z. The scope of this paper is restricted to verification techniques for Z-like specifications.

1.2 Verification of specifications

In this paper, *verification* refers to the use of mathematical analysis techniques—including proof, but not limited to it—for gaining confidence in the correctness of a specification. In general, correctness includes aptness, internal consistency, completeness, implementability, robustness, and more. Verification tasks include:

- showing that certain properties are logical consequences of the specification (including safety and liveness properties);
- syntax- and type-checking, including limiting the use of variables to the scope of their definitions;
- well-formedness checking (see below);
- proof of mathematical consistency (or satisfiability), by showing that user-introduced definitions are mathematically meaningful.

1.3 Well-formedness

We shall call an expression *well-formed* if it is syntactically (and type-) correct and it defines a value. The most common form of undefinedness in specifications involves the application of functions outside their domain: e.g.

$1/0$, $3 \bmod 0$, $\mathit{head}\ \langle\,\rangle$, $s(\#s+1)$ where $s \in \mathsf{seq}\, X$,
$\mathit{min}\ \{\,\}$, $\mathit{max}\, \mathbb{N}$, $\mu\, n{:}\mathbb{Z} \mid n^2 = 1$

Other forms of undefinedness include non-termination of recursion and cases missing from definitions.

1.4 This paper

The proof theory for the subset of VDM-SL used in this paper is that described in Bicarregui at al [2]. It agrees in spirit—if not always in detail—with Jones [12] and the emerging BSI Standard [4], and is largely supported by `mural`. The semantics for Z used for comparison in this paper is that of the draft Z Base Standard [3].

Sections 2 and 3 below provide the background needed to translate between the two notations: section 2 describes the mathematical languages and section 3 compares their specification languages. Section 4 introduces LPF and explains the treatment of partial functions. Section 5 outlines VDM's proof obligations, and section 6 compares the use of Z and VDM analysis and verification techniques on a small example specification. Section 7 sketches how to adapt the W logic for Z [3, 18] to LPF.

1.5 Other comparisons

Among other comparisons of Z and VDM, the recent paper by Hayes, Jones and Nicholls [10] offers an understanding of "the interesting differences" between the VDM and Z specification languages, together with some of the history of their development. It is written in a lively, readable form and is well recommended to readers; however, it restricts itself to expressibility of specifications and only lightly touches on verification. The response by Hall [7] presents a specifier's perspective on the comparison.

Earlier comparisons [8, 16] similarly focus mainly on notational or structural differences between the two methods. An SVRC technical report [15] compares Z and VDM verification techniques on five specification case studies, and explores the use of some other possible analysis techniques.

2 Comparison of mathematical languages

This section compares the more commonly used aspects of the VDM-SL and Z mathematical languages, with an emphasis on how VDM-SL constructs translate to their Z counterparts, so that readers familiar with Z can read the VDM notation used in other parts of the paper.

2.1 Mathematical framework

At first sight, the Z and VDM-SL notations are quite different, but for the most part the difference is superficial. The main grammatical differences are:

- Types, functions and values are distinct syntactic categories in VDM-SL but not in Z.
- Types and functions can be passed as values and returned as results in Z but not in VDM-SL.[1]
- Predicates and expressions are distinct syntactic categories in Z, whereas VDM-SL has a 'Boolean' type and treats predicates as Boolean-valued terms.

The main historical reason for these differences is intimately linked with the use of VDM as a development—not just specification—method, and is beyond the scope of this paper. Apart from the above differences, however, the constructs and primitives from one method can easily be expressed in the other method.[2]

2.2 Types

VDM types correspond to Z's declared sets. Unlike the Z approach, however, VDM makes a strict distinction between types: e.g. sets and sequences are disjoint types. Type constructors which have essentially the same meaning in the

[1] Or at least, not in that part of VDM-SL covered by the proof theory described here.
[2] e.g. Gilmore [6] describes a mechanical translation from a large subset of Z to VDM-SL.

VDM type constructor		corresponding Z term	
terminology	notation	terminology	notation
sets	X-set	finite sets	$\mathbb{F}\,X$
maps	$X \xrightarrow{m} Y$	finite (partial) functions	$X \mathrel{+\!\!\!\!\twoheadrightarrow} Y$
injective maps	$X \xleftrightarrow{m} Y$	finite injective functions	$X \rightarrowtail\!\!\!\!\!+\!\!\!\!\rightarrow Y$
sequences	X^*	sequences	$\mathrm{seq}\,X$

Table 1. Translation of VDM type constructors into Z.

two methods, but for which different terminology and/or notation are used, are shown in Table 1. This paper concentrates on data modelling using finitary (flat) types only: see Section 4.3 below for more discussion.

VDM's 'composite types' correspond roughly to Z schema types, except that fields are ordered. An example VDM composite type with its corresponding Z schema type is:

$$\begin{array}{lllll} T & :: & a & : & A \\ & & b & : & B \\ \multicolumn{5}{l}{\mathsf{inv}\ \mathit{mk}\text{-}T(x,y) \ \triangleq\ \mathit{Tinv}(x,y)} \end{array}$$

```
┌ T ──────────────────────
│ a: A
│ b: B
├──────────
│ Tinv(a, b)
└─────────────────────────
```

2.3 Functions and values

The two methods have many value constructs in common. Constructs with essentially the same meaning but different notation are summarized in Table 2. Note that VDM has Boolean terms true and false, and a polymorphic conditional construct

$$\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_ : \mathbb{B} \times A \times A \to A$$

Composite type definitions give rise to constructor and selector functions. For example, the definition T above introduces a constructor function $\mathit{mk}\text{-}T$ which is a partial function from $A \times B$ to T with domain defined by Tinv. The value of $\mathit{mk}\text{-}T(x,y)$ thus corresponds to the Z binding $\langle\!|\ a \leadsto x, b \leadsto y\ |\!\rangle$. Z and VDM use the same notation for selector functions.

The two methods use different notation for set comprehensions: VDM writes $\{f(x) \mid x{:}\,A \cdot P(x)\}$ where Z writes $\{x{:}\,A \mid P(x) \bullet f(x)\}$.

3 Comparison of specification notations

In model-oriented styles, a system specification consists of a (user-defined) data model, a state space and state transitions. Data models are specified by defining types and values, together with auxiliary functions and predicates on those types. (Auxiliary constructs are ones which are introduced for the purposes of modelling, simplification or explanation of a specification, but which are not intended to be implemented.)

construct	VDM	Z	construct	VDM	Z
first	fst p	$\mathit{first}\ p$	empty sequence	$[\,]$	$\langle\,\rangle$
second	snd p	$\mathit{second}\ p$	sequence head	hd s	$\mathit{head}\ s$
set cardinality	card s	$\#s$	sequence tail	tl s	$\mathit{tail}\ s$
finite power set	$\mathcal{F} s$	$\mathbb{F}\ s$	concatenation	$s_1 \frown s_2$	$s_1 \frown s_2$
number range	$\{i, \ldots, j\}$	$i \mathrel{..} j$	distributed concat	conc ss	$\frown/\ ss$
empty map	$\{\mapsto\}$	$\{\,\}$	sequence indices	inds s	dom s
map range	rng m	ran m	sequence elements	elems s	ran s
map overwrite	$m_1 \dagger m_2$	$m_1 \oplus m_2$	sequence length	len s	$\#s$

Table 2. Some differences in notation for value constructors.

3.1 Types

Types are defined using the mathematical notation of Section 2.2 above. In VDM-SL, a 'type invariant' can be used to restrict membership of a type: e.g.

$$\mathit{IncreasingSeqs} = \mathbb{N}^*$$

$$\mathsf{inv}\ s \quad \triangleq \quad \forall i{:}\mathbb{N} \cdot 1 \leq i < \mathsf{len}\, s \Rightarrow s(i) < s(i+1)$$

Z typically uses schema inclusion or set comprehension to achieve the same effect.

Recursive type definitions are allowed in VDM; however, in order to reason effectively about such definitions, it is necessary to have corresponding induction rules and as yet no fully general means has been developed for generating these.[3] In Z, recursive type definitions can only be made via free types.

3.2 Auxiliary functions

In VDM-SL, user-introduced functions can be defined in two ways:

1. *directly*, by giving an expression whose value is the result returned by the function, or
2. *indirectly*, in terms of pre- and post-conditions.

By contrast, Z uses 'axiomatic definitions', consisting of a 'declared set' (roughly, the function's signature) and a predicate which defines the function. A VDM direct definition of the form

$$\mathit{factorial} : \mathbb{Z} \to \mathbb{Z}$$

$$\mathit{factorial}(n) \quad \triangleq \quad \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * \mathit{factorial}(n-1)$$

$$\mathsf{pre}\ n \geq 0$$

[3] cf. §13.4 of Bicarregui et al [2].

corresponds to a Z axiomatic definition of the form

$$\begin{array}{|l}
\mathit{factorial}: \mathbb{Z} \pfun \mathbb{Z} \\
\hline
\mathsf{dom}\,\mathit{factorial} = \{n: \mathbb{Z} \mid n \geq 0\} \\
\mathit{factorial}(0) = 1 \\
\forall\, n: \mathbb{Z} \bullet n > 0 \Rightarrow \mathit{factorial}(n) = n * \mathit{factorial}(n-1)
\end{array}$$

Note that, in VDM, the domain of a partial function is defined by stating a precondition for that function in a pre clause. (Functions without explicit preconditions are assumed to be total.) Note also that recursive definitions are allowed.

A VDM indirect definition of the form

$$\begin{array}{l}
\mathit{sqroot}\ (n: \mathbb{N})\ r: \mathbb{N} \\
\mathsf{post}\ r * r \leq n < (r+1) * (r+1)
\end{array}$$

corresponds to a Z axiomatic definition of the form

$$\begin{array}{|l}
\mathit{sqroot}: \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall\, n: \mathbb{N} \bullet \exists\, r: \mathbb{N} \bullet \\
\quad r = \mathit{sqroot}(n) \\
\quad r * r \leq n < (r+1) * (r+1)
\end{array}$$

3.3 States

The 'state' of the system being specified is defined in a state environment in VDM-SL; the state schema is not formally distinguished from other schemas in Z. By way of example, Fig. 1 shows a VDM state definition for a simple Directed Acyclic Graph (DAG) system, together with its auxiliary functions. Initially the graph is empty (has no nodes). The constraints—including the requirement that the graph be acyclic—are written in an invariant (inv) clause. The initial condition, defining the set of all possible initial states, is written in the init clause. The auxiliary function *has_path* checks whether there is a path through the graph from one node to another; *has_no_circs* checks that there are no circular paths.

A corresponding Z specification is given in Fig. 2. The Z specification is more concise in this case—as in many cases—because it is better able to take advantage of the Z mathematical toolkit.

3.4 State transitions

State transitions are modelled as 'operations' in VDM. Operation specifications can involve input and/or output variables. An 'external variables' (ext) clause (or 'frame') says which state variables are used in or affected by the operation: rd variables are read-only; wr variables are read/write; unmentioned variables do not change value. This corresponds roughly to Z's Δ and Ξ conventions.

$$
\begin{array}{l}
\mathsf{state}\ DAG\ \mathsf{of} \\
\qquad\qquad nodes\colon X\text{-}\mathsf{set} \\
\qquad edges\colon (X \times X)\text{-}\mathsf{set} \\
\mathsf{inv}\ mk\text{-}DAG(V, E)\ \ \triangleq \\
\quad (\forall x, y\colon X \cdot (x, y) \in E \Rightarrow \\
\qquad x \in V \wedge y \in V) \\
\quad \wedge\ has\_no\_circs(E) \\
\mathsf{init}\ dag\ \ \triangleq\ \ dag.nodes = \{\,\} \\
\mathsf{end}
\end{array}
\qquad
\begin{array}{l}
has\_no\_circs : (X \times X)\text{-}\mathsf{set} \to \mathbb{B} \\
has\_no\_circs(E)\ \ \triangleq \\
\quad \forall x\colon X \cdot \neg\, has\_path(x, x, E) \\
\\
has\_path : X \times X \times (X \times X)\text{-}\mathsf{set} \to \mathbb{B} \\
has\_path(x, y, E)\ \ \triangleq\ \ \exists ps\colon X^{*} \cdot \\
\qquad \mathsf{len}\, ps > 1 \wedge ps(1) = x \\
\qquad \wedge\ ps(\mathsf{len}\, ps) = y \\
\qquad \wedge\ (\forall k\colon \mathbb{N}_1 \cdot k < \mathsf{len}\, ps \Rightarrow \\
\qquad\quad (ps(k), ps(k+1)) \in E)
\end{array}
$$

Fig. 1. Part of the data model for DAG in VDM.

$$
\begin{array}{|l}
paths\colon (X \leftrightarrow X) \to (X \leftrightarrow X) \\
\hline
\forall\, E\colon X \leftrightarrow X \bullet paths(E) = E^{+}
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{DAG} \\
\hline
nodes\colon \mathbb{F}\, X \\
edges\colon X \leftrightarrow X \\
\hline
edges \subseteq nodes \times nodes \\
edges \in has\_no\_circs \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
has\_no\_circs\colon \mathbb{P}\,(X \leftrightarrow X) \\
\hline
has\_no\_circs = \\
\quad \{E\colon X \leftrightarrow X \mid \\
\qquad \forall\, x\colon X \bullet (x, x) \notin paths(E)\}
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{DAG_{Init}} \\
\hline
DAG \\
\hline
nodes = \{\,\} \\
\hline
\end{array}
$$

Fig. 2. The corresponding DAG data model in Z.

As with functions, both direct and indirect definitions of operations are possible in VDM-SL, although direct definitions use an algorithmic subset of VDM-SL which is not covered in this paper. Indirect definitions are stated in terms of pre- and post-conditions. The 'precondition' (pre) is a formula involving input variables and state variables from the externals clause only; the effect of an operation is defined only on those values which satisfy the precondition. The 'postcondition' (post) is a formula involving input/output variables and state variables from the externals clause. For writable state variables in the postcondition (only), a hook ($\overleftarrow{\_}$) indicates the variable's value before the operation is applied; the unadorned variable denotes its value afterwards.[4] This contrasts with the Z convention of

[4] Note that the hook is not used in the precondition.

using a primed variable for the post-value and an unadorned variable for the pre-value.

$AddEdge\ (x\colon X,\, y\colon X)$
ext rd $nodes\colon X$-set
    wr $edges\colon (X \times X)$-set
pre $\{x, y\} \subseteq nodes$
    $\wedge\ candidate(x, y, edges)$
post $edges = \overleftarrow{edges} \cup \{(x, y)\}$

$candidate : X \times X \times (X \times X)\text{-set} \to \mathbb{B}$

$candidate(x, y, E) \ \triangleq$
    $x \neq y \wedge \neg\, has\_path(y, x, E)$

Fig. 3. A VDM operation for adding an edge to the DAG, with an auxiliary function.

Fig. 3 shows a VDM operation for adding an edge to the DAG, together with its auxiliary function. The operation's precondition ensures that adding the edge will not create a circularity. (In Section 5 below there is a proof obligation to show that the stated precondition is strong enough to ensure that the postcondition is achievable.) Fig. 4 shows a corresponding Z specification.

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

$AddEdge$
ΔDAG
$x?, y?\colon X$

$nodes' = nodes$
$edges' = edges \cup \{(x?, y?)\}$

Fig. 4. The Z operation for adding an edge to the DAG.

3.5 Discussion

The Z schema notation is a rich, flexible notation for structuring specifications which has no equal in VDM.

In Z, schemas are used for many different purposes, including for the definition of the state, types and operations, as well as for expressing auxiliary concepts. Schema inclusion is a very effective technique not supported in VDM.

The *schema calculus* is used for defining new schemas in terms of old ones: for example, in Fig. 5 the schema *IsNode* is an auxiliary schema, used to simplify the definition of other schemas—there is no intention that it be implemented.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{IsNode}} \\
\hline
\mathit{DAG} \\
x?: X \\
\hline
x? \in \mathit{nodes} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InsertNode}} \\
\hline
\Delta \mathit{DAG} \\
x?: X \\
\hline
\mathit{nodes}' = \mathit{nodes} \cup \{x?\} \\
\mathit{edges}' = \mathit{edges} \\
\hline
\end{array}$$

$$\mathit{AddNode} \mathrel{\widehat{=}} \neg\, \mathit{IsNode} \wedge \mathit{InsertNode}$$

Fig. 5. The Z operation for adding a node to the DAG.

On the other hand, a VDM specification has more formal structure than a Z specification, in the sense that it makes a formal distinction between state definitions, types and operations, and it insists that preconditions be given explicitly for partial functions and partial operations.

4 The Logic of Partial Functions

This section discusses the logic underlying VDM: the *Logic of Partial Functions* (LPF) [1, 2, 5, 14].

4.1 Motivation

Classical treatments of logic differ in how they handle undefined terms such as $\mathsf{hd}\,[\,]$, and many treatments simply ignore them. One common approach is to assign them an "arbitrary" value and to be careful that nothing non-trivial can be deduced about that value: for example, in the deductive system for Z in the Z Base Standard [3] all well-typed terms are assumed to denote values. By contrast, LPF was developed as a logic for models in which undefined terms do *not* denote values.

4.2 The formal logic

Formally, LPF differs from most classical treatments of logic in four main ways:

1. Formulas (propositions) are treated as just another kind of term (namely, Boolean-valued terms). Predicates are treated as Boolean-valued functions.
2. Only fully defined, well-typed terms denote values.

3. Propositions may not define a truth value (i.e., a proposition may be neither 'true' nor 'false').
4. An assertion '$a: A$' means that the term a is fully defined and denotes a value of type A.

A term is said to be *well-formed* if it denotes a value. It is called a *well-formed formula* if it denotes a Boolean value.

An example of an undefined proposition is the equation $1/0 = 2/0$. On the other hand, the formula

$$s \neq [\,] \Rightarrow \mathsf{hd}\ s \in \mathsf{elems}\ s$$

is true for all sequences s, even though $\mathsf{hd}\ [\,]$ is not defined. To accommodate the presence of undefined subterms in expressions, the semantics of the propositional connectives are "extended" in LPF, as indicated in the following truth tables:

$\wedge$	true	false	$\perp$
true	true	false	$\perp$
false	false	false	false
$\perp$	$\perp$	false	$\perp$

$\Rightarrow$	true	false	$\perp$
true	true	false	$\perp$
false	true	true	true
$\perp$	true	$\perp$	$\perp$

where $\perp$ represents undefinedness.

Since in LPF a proposition may be neither true nor false, it follows that not all of the laws of classical logic are valid in LPF. The converse does hold however: that is, *all laws of LPF are valid classic laws.* Fig. 6 shows some of the laws which are common to both logics, presented in Natural Deduction style.[5]

$$\frac{P \wedge Q}{P} \qquad \frac{P \wedge Q}{Q} \qquad \frac{P,\ Q}{P \wedge Q} \qquad \frac{P,\ \neg P}{Q} \qquad \frac{P,\ P \Rightarrow Q}{Q}$$

$$\frac{P}{P \vee Q} \qquad \frac{Q}{P \vee Q} \qquad \frac{P \vee Q \quad \begin{array}{c}[P]\\ R\end{array} \quad \begin{array}{c}[Q]\\ R\end{array}}{R}$$

Fig. 6. Laws common to LPF and classical logic.

An example of a classical law which is not valid in LPF is the law of "Excluded Middle"

$$\frac{}{P \vee \neg P}$$

[5] In `mural`, the Natural Deduction approach was carried through to full formality. An important innovation was to allow scoping of variables within subproofs ("blocks"): see §1.3 of [2].

which does not hold in LPF when P is an undefined proposition. Typically, however, the laws of classical logic can be converted into valid LPF laws by adding hypotheses to ensure their conclusions are well-formed. For example, the LPF version of "Excluded Middle" is:

$$\frac{P\colon \mathbb{B}}{P \vee \neg P}$$

(In other words, $P \vee \neg P$ is true provided P denotes the value 'true' or 'false'.)

In practice, this means that LPF has a number of different laws where classically a single law would suffice. For example, there are two LPF laws for $\wedge$-formation:

$$\frac{P\colon \mathbb{B} \quad \begin{matrix}[P]\\ Q\colon \mathbb{B}\end{matrix}}{(P \wedge Q)\colon \mathbb{B}} \qquad \frac{Q\colon \mathbb{B} \quad \begin{matrix}[Q]\\ P\colon \mathbb{B}\end{matrix}}{(P \wedge Q)\colon \mathbb{B}}$$

The first says that $P \wedge Q$ is a well-formed formula if P is a well-formed formula and, assuming P is true, Q is a well-formed formula. This rule accounts for the well-formedness of, for example, the term '$s \neq [\,] \wedge \mathsf{hd}\, s \in \mathsf{elems}\, s$'. The second law covers the symmetric case.

A more subtle example is the "Deduction Theorem", which is stated classically as follows:

$$\frac{\begin{matrix}[P]\\ Q\end{matrix}}{P \Rightarrow Q}$$

(In other words, if Q can be proven by assuming P, then $P \Rightarrow Q$ is true.) The semantics of LPF say that it is possible to deduce anything from an undefined proposition P, but that $P \Rightarrow Q$ is undefined if P is undefined and Q is false. Thus, the LPF version of the Deduction Theorem needs an extra hypothesis to ensure that P is a defined proposition: *viz.*

$$\frac{P\colon \mathbb{B} \quad \begin{matrix}[P]\\ Q\end{matrix}}{P \Rightarrow Q}$$

In LPF, quantified variables range over defined values of a type only, so the laws for quantifiers are the same as for (typed) classical logic:

$$\frac{\begin{matrix}[x\colon A]\\ P(x)\end{matrix}}{\forall y\colon A \cdot P(y)} \qquad \frac{a\colon A,\ \forall x\colon A \cdot P(x)}{P(a)}$$

Equality is 'strict', so the law of reflexivity of equality requires a well-formedness hypothesis:

$$\frac{a\colon A}{a = a}$$

$$\frac{}{\{\}: A\text{-set}} \qquad \frac{a: A,\ s: A\text{-set}}{add(a,s): A\text{-set}} \qquad \frac{a: A}{a \notin \{\}} \qquad \frac{a: A,\ b: A,\ s: A\text{-set}}{b \in add(a,s) \ \Leftrightarrow\ b = a \vee b \in s}$$

$$\frac{P(\{\}) \qquad \begin{array}{c}[a: A,\ s: A\text{-set},\ a \notin s,\ P(s)]\\ P(add(a,s))\end{array}}{\forall s: A\text{-set} \cdot P(s)}$$

Fig. 7. LPF laws for sets.

Some other laws of LPF are given in Figs. 7 and 8.[6]

Note that the assertion $a: A$ implies that a denotes a value of type A, and in particular that a satisfies any invariants that might be associated with A (e.g. when A is a composite type). This very strict interpretation of typing assertions means that type-checking is undecidable in this logic.

$$\frac{P: \mathbb{B} \quad \begin{array}{c}[P]\\ Q: \mathbb{B}\end{array}}{(P \Rightarrow Q): \mathbb{B}} \qquad \frac{\begin{array}{c}[x: A]\\ P(x): \mathbb{B}\end{array}}{(\forall y: A \cdot P(y)): \mathbb{B}} \qquad \frac{a: A,\ f: A \xrightarrow{m} B,\ a \in \mathsf{dom}\, f}{f(a): B}$$

Fig. 8. Some LPF laws for well-formedness and type-checking.

4.3 Finiteness

This paper concentrates on data modelling using *finite* values only: i.e., finite sets rather than possibly-infinite sets, and finite functions rather than total functions. Partly our reasons are foundational, and for example concern the non-computability of equality between infinite values.[7] But partly also they are practical: for example, it is much easier to reason in the theory of finite sets than in, say, Zermelo-Frankel Set Theory, since the former has a simple Induction Principle (cf. Fig. 7).

As it happens, the restriction to finite values is often not overly constraining for the specifier.[8] In practice, it often simply means using finite power sets ($\mathbb{F}$) rather than arbitrary power sets ($\mathbb{P}$), and finite functions ($\twoheadrightarrow$) rather than total ($\rightarrow$) or partial ($\nrightarrow$) functions.

[6] Note that $add(a,s) \ \triangleq\ s \cup \{a\}$.

[7] See Chapter 13 of Bicarregui et al [2].

[8] Hodges [11] argues that all the specifications in Hayes [9] can be rewritten so that they use (hereditarily) finite sets only.

5 Verification in VDM

In the `mural` approach to specification verification, each VDM specification has a corresponding mathematical theory, in which the definitions of the specification are represented as axioms and definitions of the theory. Type-, well-formedness and satisfiability checking are carried out by proving that certain theorems (called *proof obligations*) are logical consequences of the axioms and definitions of the theory. The mathematical theory and proof obligations are generated by mechanical translation from the specification.

This section briefly outlines the proof obligations for VDM specifications. Figures 5–13 sketch the axioms and proof obligations corresponding to each of the main specification constructs discussed in Section 3 above.

The axioms are formulated in such a way as to adhere to the strictures of LPF: enough hypotheses are given to ensure that the conclusions are valid, without assuming in advance that the definitions are well-formed. Thus, for example, the axiom which introduces a direct function (see Fig. 10) has hypotheses which ensure that the arguments are well-formed and of the correct type, that the precondition holds, and that the body of the definition is well-formed and of the correct type. There are separate proof obligations to show that the precondition is a well-formed formula, and that the body is well-formed and of the correct type. Once the proof obligations have been established, it is easy to derive a 'working version' of the introduction rule for the function, in the form

$$\frac{x\colon A,\ \ y\colon B,\ \ \mathit{fpre}(x,y)}{f(x,y) = \mathit{fdef}(x,y)}$$

5.1 Well-formedness

The role of well-formedness proof obligations is to check that definitions are *mathematically complete*. The well-formedness proof obligations for the different specification constructs can be paraphrased as follows:

type definitions: the invariant is a well-formed formula.

direct function definitions: the precondition is a well-formed formula; the body is well-formed and of the correct type.

indirect function definitions: the pre- and post-conditions are well-formed formulae. (Note that the postcondition is only required to be well-formed on the function's domain: i.e., the precondition can be assumed when proving that the postcondition is well-formed.)

state definitions: the state invariant and initial condition are well-formed formulae. (The state invariant can be assumed when dealing with the initial condition.)

operation definitions: the pre- and post-conditions are well-formed formulae. (The state invariant can be assumed in both cases, and the precondition can be assumed when dealing with the postcondition.)

Pattern

$T = A$

inv $t \;\; \triangleq \;\; Tinv(t)$

Axioms

$$\frac{t{:}\,A,\;\; Tinv(t)}{t{:}\,T} \qquad \frac{t{:}\,T}{t{:}\,A} \qquad \frac{t{:}\,T}{Tinv(t)}$$

Proof obligations

$$\frac{t{:}\,A}{Tinv(t){:}\,\mathbb{B}}$$

Pattern

$f : A \times B \to R$

$f(x, y) \;\; \triangleq \;\; fdef(x, y)$

pre $fpre(x, y)$

Axiom

$$\frac{\begin{array}{c} x{:}\,A,\;\; y{:}\,B, \\ fpre(x, y), \\ fdef(x, y){:}\,R \end{array}}{f(x, y) = fdef(x, y)}$$

Proof obligations

$$\frac{x{:}\,A,\;\; y{:}\,B}{fpre(x, y){:}\,\mathbb{B}} \qquad \frac{x{:}\,A,\;\; y{:}\,B,\; fpre(x, y)}{fdef(x, y){:}\,R}$$

Fig. 9. Axioms and proof obligations for a type definition T.

Fig. 10. Axioms and proof obligations for a direct function definition f.

Under the LPF approach, the presence of mathematical incompletenesses in the definitions of a specification is revealed by the inability to discharge a well-formedness proof obligation. The point where the proof stalls is typically the point where the undefinedness occurs, and consideration of how to make progress in the proof often shows how to fix the specification. This is illustrated in numerous examples on a sizeable case study in Chapter 12 of [2].

5.2 Satisfiability

The other kind of proof obligation is called a *satisfiability* proof obligation: it checks that an implicit definition is *mathematically consistent* (or mathematically meaningful), in the sense that there exists a mathematical object which satisfies the definition. Satisfiability of axiomatic definitions (cf. implicit function definitions below) is a necessary prerequisite for a data model to be mathematically consistent: e.g. using

$$\forall\, s{:}\,\mathbb{F}\,\mathbb{N} \setminus \{\{\,\}\} \bullet \exists\, m{:}\,\mathbb{N} \bullet max\ s \leq m$$

it is possible to derive a contradiction from the following ill-declared definition:

$$\begin{array}{|l} evens{:}\,\mathbb{F}\,\mathbb{N} \\ \hline evens = \{n{:}\,\mathbb{N} \mid n \bmod 2 = 0\} \end{array}$$

Pattern

$g\ (x\colon A, y\colon B)\ r\colon R$
pre $gpre(x, y)$
post $gpost(x, y, r)$

Axiom

$$\frac{x\colon A,\ \ y\colon B,\\ gpre(x,y),\\ \exists r\colon R \cdot gpost(x,y,r)}{g(x,y)\colon R \wedge gpost(x,y,g(x,y))}$$

Proof obligations

$$\frac{x\colon A,\ \ y\colon B}{gpre(x,y)\colon \mathbb{B}} \qquad \frac{x\colon A,\ \ y\colon B,\ \ r\colon R,\\ gpre(x,y)}{gpost(x,y,r)\colon \mathbb{B}}$$

$$\frac{x\colon A,\ \ y\colon B,\\ gpre(x,y)}{\exists r\colon R \cdot gpost(x,y,r)}$$

Pattern
state S of
  $a\colon A$
  $b\colon B$
  $c\colon C$
inv $mk\text{-}S(x, y, z) \ \ \triangleq\ \ Sinv(x, y, z)$
init $s \ \ \triangleq\ \ init(s)$
end

Axioms

$$\frac{x\colon A,\ \ y\colon B,\ \ z\colon C,\\ Sinv(x,y,z)}{mk\text{-}S(x,y,z)\colon S}$$

etc, as for composite types

Proof obligations

$$\frac{x\colon A,\ \ y\colon B,\ \ z\colon C}{Sinv(x,y,z)\colon \mathbb{B}} \qquad \frac{s\colon S}{init(s)\colon \mathbb{B}}$$

$$\frac{}{\exists s\colon S \cdot init(s)}$$

Fig. 11. Axioms and proof obligations for an indirect function definition g.

Fig. 12. Axioms and proof obligations for a state definition S.

The satisfiability proof obligations for specification components can be paraphrased as follows:

indirect function definitions: for all arguments in the domain of the function, there is at least one corresponding result which satisfies the postcondition.
state definitions: the state invariant is satisfiable (so the state space is non-empty); if an initial condition has been given, it should be satisfiable.
operation definitions: if the operation is enabled for a given input and state of the system, then there exists a transition to some other state, with an appropriate output value, which satisfies the operation's specification.

Failure to discharge a satisfiability proof obligation often reveals hidden assumptions about the conditions under which the operation will be invoked.

Pattern

$OP\ (i{:}\, I)\ r{:}\, R$
ext rd $a{:}\, A$
  wr $b{:}\, B$
pre $opre(i, a, b)$
post $opost(i, r, a, \overleftarrow{b}, b)$

Proof obligations

$$\frac{i{:}\, I,\ \ mk\text{-}S(x, y, z){:}\, S}{opre(i, x, y){:}\, \mathbb{B}}$$

$$\frac{\begin{array}{c} i{:}\, I,\ \ r{:}\, R, \\ mk\text{-}S(x, \overleftarrow{y}, z){:}\, S,\ \ mk\text{-}S(x, y, z){:}\, S, \\ opre(i, x, \overleftarrow{y}) \end{array}}{opost(i, r, x, \overleftarrow{y}, y){:}\, \mathbb{B}}$$

$$\frac{\begin{array}{c} i{:}\, I,\ \ mk\text{-}S(x, \overleftarrow{y}, z){:}\, S, \\ opre(i, x, \overleftarrow{y}) \end{array}}{\begin{array}{c} \exists r{:}\, R, y{:}\, B \cdot Sinv(x, y, z) \wedge \\ opost(i, r, x, \overleftarrow{y}, y) \end{array}}$$

Fig. 13. Axioms and proof obligations for an operation specification *OP*.

Note that if the operation's precondition has been derived by calculating the weakest precondition [17], then that calculation can serve as a proof of satisfiability.

5.3 Other proof opportunities

To the above satisfiability proof obligations could be added various "proof opportunities"—statements which would normally be true, but which are not strictly required by the methodology. For example, since it would not normally be the specifier's intention that a defined type is empty, there is an opportunity to show that each defined type is inhabited (i.e., its invariant is satisfiable):

$\exists t{:}\, A \cdot Tinv(t)$

Similarly, defined functions should be non-trivial (i.e., their preconditions should be satisfiable):

$\exists x{:}\, A, y{:}\, B \cdot fpre(x, y)$
$\exists x{:}\, A, y{:}\, B \cdot gpre(x, y)$

For each operation, there should be at least one set of circumstances under which the operation is enabled (i.e., its precondition should be satisfiable):

$\exists i{:}\, I,\, mk\text{-}S(x, y, z){:}\, S \cdot opre(i, x, y)$

Wordsworth [19] (p.259) gives two proof obligations for Z specifications: consistency of the state, which corresponds to satisfiability of the state definition above; and "implementability of operations", which corresponds to the satisfiability proof opportunity for operations above.

6 An example

This section illustrates some of the main points of the above sections on a simple specification. First, a simple case study is described informally and a naive Z specification is given. The requirements are then formalized in VDM, and the effect of the restriction to finite values is noted. Finally, the analysis and verification techniques from Sections 4 and 5 are applied.

The case study concerns a simple Traffic Management System (TMS). The TMS is to manage a traffic region which is divided into a number of individual zones. Each zone has a defined limit to the number of vehicles it can contain (its *capacity*). Among the operations to be specified is one for moving a vehicle from one zone to another, subject to the safety property that zones' capacities are not exceeded. There would be other operations for adding new vehicles to the system, changing the configuration of zones, and so on, but they will not be considered here.

6.1 A Z analysis

A first attempt to specify the TMS in Z is shown in Fig. 14. Capacity information is modelled as a (total) function from zones to natural numbers. Location of vehicles is modelled as a function from vehicles to zones. The safety property is stated as a constraint on the state space. The operation moves vehicle $v?$ into zone $z?$.

Calculation reveals the weakest precondition to be

```
┌─ pre MoveVehicle ─────────────────────────────────────
│ TMS
│ v?: Vehicles;  z?: Zones
├──────────────────────
│ location(v?) ≠ z? ⇒ #location~{z?} < capacity(z?)
└───────────────────────────────────────────────────────
```

Equipped with such knowledge, we might well question whether the operation should be allowed to "move" a vehicle into a zone it already occupies; if not, then $location(v?) \neq z?$ should be added to the schema body.

Turning to the analysis of Z specifications suggested by Wordsworth [19], in order to show the state schema is satisfiable, it is necessary to give values for the *location* and *capacity* which satisfy the schema predicate. To be able to actually

$[Zones, Vehicles]$

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

TMS

$capacity\colon Zones \rightarrow \mathbb{N}$

$location\colon Vehicles \rightarrow Zones$

$\forall z\colon Zones \bullet$

$\quad \# location^{\sim}\{z\} \leq capacity(z)$

MoveVehicle

ΔTMS

$v?\colon Vehicles;\ \ z?\colon Zones$

$capacity' = capacity$

$location' = location \oplus \{v? \mapsto z?\}$

Fig. 14. A naive Z specification of the TMS.

exhibit a value of the state satisfying the invariant, it would be necessary to postulate the existence of an infinite set of distinguished zones (say $z_0, z_1, \ldots$) and an enumeration of the vehicles (say $v_0, v_1, \ldots$); then one could give e.g. the following term as a witness value:

$\langle\!| \, capacity \leadsto \lambda\, z\colon Zones \bullet 1, location \leadsto \{i\colon \mathbb{N} \bullet v_i \mapsto z_i\} \, |\!\rangle$

Similarly, the *MoveVehicle* operation is "implementable" provided there is a state in which some zone is under capacity. These proof obligations seem to do little to further our understanding of the system being specified.

6.2 A VDM analysis

We now specify the TMS in VDM, modelling the state variables as finite functions of types $Zone \xrightarrow{m} \mathbb{N}$ and $Vehicle \xrightarrow{m} Zone$ (Fig. 15). Since the expression which finds the number of vehicles in a given zone is relatively complicated in VDM-SL, we have decided to encapsulate it as an auxiliary function *content*. Formally, the proof obligation to show well-formedness of *content* is

$$\frac{z\colon Zone,\ loc\colon Vehicle \xrightarrow{m} Zone}{(\mathsf{card}\,(\mathsf{dom}\,(loc \rhd \{z\})))\colon \mathbb{N}}$$

The constraint that zone z's capacity is not exceeded can be stated as $content(z, loc) \leq cap(z)$, where loc is the value of the location mapping and cap is the value of the capacity mapping. But since cap is a finite function, $cap(z)$ will be defined for only finitely many zones z. We must ask ourselves, for which z are we interested in expressing the constraint? A reasonable answer would be to restrict to occupied zones (i.e., $z \in \mathsf{rng}\, loc$) and to state the constraint as

$$\forall z\colon Zone \cdot z \in \mathsf{rng}\, loc \Rightarrow content(z, loc) \leq cap(z)$$

To be able to prove that this formula is well-formed, however, it is necessary to ensure that $cap(z)$ is defined for all $z \in \mathsf{rng}\, loc$, or in other words, that $\mathsf{rng}\, loc \subseteq \mathsf{dom}\, cap$. This constraint has been added to the state invariant in

$$\begin{array}{l}
\mathsf{state}\ \mathit{TMS}\ \mathsf{of} \\
\qquad\qquad \mathit{capacity}: \mathit{Zone} \xrightarrow{m} \mathbb{N} \\
\qquad \mathit{location}: \mathit{Vehicle} \xrightarrow{m} \mathit{Zone} \\
\mathsf{inv}\ \mathit{mk\text{-}TMS}(\mathit{cap}, \mathit{loc}) \ \ \triangleq \\
\quad \mathsf{rng}\ \mathit{loc} \subseteq \mathsf{dom}\ \mathit{cap}\ \wedge \\
\quad \forall z: \mathit{Zone} \cdot z \in \mathsf{rng}\ \mathit{loc} \Rightarrow \\
\qquad\quad \mathit{content}(z, \mathit{loc}) \leq \mathit{cap}(z) \\
\mathsf{end} \\
\\
\mathit{content} : \mathit{Zone} \times (\mathit{Vehicle} \xrightarrow{m} \mathit{Zone}) \\
\qquad\quad \rightarrow \mathbb{N} \\
\\
\mathit{content}(z, \mathit{loc}) \ \ \triangleq \\
\quad \mathsf{card}\ (\mathsf{dom}\ (\mathit{loc} \rhd \{z\}))
\end{array}$$

$$\begin{array}{l}
\mathit{MoveVehicle}\ (v: \mathit{Vehicle}, z: \mathit{Zone}) \\
\mathsf{ext\ rd}\ \mathit{capacity}: \mathit{Zone} \xrightarrow{m} \mathbb{N} \\
\quad \mathsf{wr}\ \mathit{location}: \mathit{Vehicle} \xrightarrow{m} \mathit{Zone} \\
\mathsf{pre}\ v \in \mathsf{dom}\ \mathit{location}\ \wedge \\
\quad z \in \mathsf{dom}\ \mathit{capacity}\ \wedge \\
\quad \mathit{content}(z, \mathit{location}) < \mathit{capacity}(z) \\
\mathsf{post}\ \mathit{location} = \overleftarrow{\mathit{location}} \dagger \{v \mapsto z\}
\end{array}$$

Fig. 15. The TMS specification in VDM.

Fig. 15. The proof that the full state invariant is well-formed is straightforward. The state invariant is satisfiable for example when $\mathit{cap} = \mathit{loc} = \{\mapsto\}$.

Now let us turn our attention to the operation for moving a vehicle v into zone z. Formally, the satisfiability proof obligation for *MoveVehicle* is

$$\frac{\begin{array}{c} v: \mathit{Vehicle},\ z: \mathit{Zone}, \mathit{mk\text{-}TMS}(\mathit{cap}, \overleftarrow{\mathit{loc}}): \mathit{TMS}, \\ v \in \mathsf{dom}\ \overleftarrow{\mathit{loc}} \wedge z \in \mathsf{dom}\ \mathit{cap} \wedge \mathit{content}(z, \overleftarrow{\mathit{loc}}) < \mathit{cap}(z) \end{array}}{\exists \mathit{loc}: \mathit{Vehicle} \xrightarrow{m} \mathit{Zone} \cdot \mathit{inv\text{-}TMS}(\mathit{cap}, \mathit{loc}) \wedge \mathit{loc} = \overleftarrow{\mathit{loc}} \dagger \{v \mapsto z\}}$$

where $\mathit{inv\text{-}TMS}(\mathit{cap}, \mathit{loc})$ is the state invariant. For the new value of *location* to be consistent with the state invariant, we need at least that

$$\mathsf{rng}\,(\overleftarrow{\mathit{loc}} \dagger \{v \mapsto z\}) \subseteq \mathsf{dom}\ \mathit{cap}$$

and hence that $z \in \mathsf{dom}\ \mathit{cap}$, or in other words that z is one of the known zones. With experience, the specifier would also question whether vehicle v is already known to the system ($v \in \mathsf{dom}\ \mathit{loc}$) or whether it is a "new" vehicle; let us assume the former. Finally, to achieve the safety property, the destination zone z must have room to accomodate v, so a statement saying that z is below capacity has been added to the precondition.

The crux of the satisfiability proof is in showing that the state invariant holds for the new value of the *location* variable, but this is quite straightforward, given the operation's precondition. Detailed examination of the proof would show that the precondition is stronger than it needs to be: e.g. the assumption that z is under capacity is not required in the case when $\overleftarrow{\mathit{loc}}(v) = z$. Just as in the Z analysis, one is lead to question whether the operation should be allowed to "move" a vehicle into the zone it is already occupying.

6.3 Discussion

It is dangerous to draw too many conclusions from a single small example. The requirements analysis performed in formulating the Z specification was rather superficial, and did little more than write down—albeit more formally—the requirements from the case study's description. Of course, the results depend to some degree on modelling decisions, and many experienced Z specifiers would undoubtedly take the analysis a lot further than illustrated above. But the example illustrates that partiality analysis has a role to play in the analysis of specifications. In this case, it revealed an implicit requirement which may not have been obvious in the informal specification of the problem: namely, that all occupied zones have defined capacities.

The example illustrates how the combination of LPF and the techniques of Section 5 can reveal hidden deeper relationships between different parts of the specification, which in turn can reveal hidden assumptions or previously undisclosed requirements.

7 Adapting W for LPF

The above techniques could easily be incorporated into a Z methodology as a discipline, without necessarilly formalizing the analysis. For those readers who might be interested in how to formalize partiality analysis for Z specifications, however, this section contains some notes on adapting W [3, 18] for LPF.

It is first necessary to weaken some of the basic rules of W to make them consistent with a new underlying semantic model in which the meaning of expressions and predicates may be undefined. A sequent

$d \mid p_1, \ldots, p_m \vdash q_1, \ldots, q_m$

would be *valid* in the new interpretation if, whenever the environment is enriched with the declarations of d in such a way that the property of d and the predicates $p_1, \ldots, p_m$ denote the truth value true, then at least one of the q_i denotes true.

Under this new interpretation, the W rule for negation introduction

$$\frac{P \vdash}{\vdash \neg P}$$

would not be valid when P is undefined. The rule needs an additional hypothesis to the effect that P is defined. One solution would be to add a Boolean type to Z and to use the typing approach from above (i.e., add $P : \mathbb{B}$ as hypothesis). Another solution, requiring a less radical change to Z, would be to represent the fact that P is defined by the assertion δP, where $\delta P == \neg(P \land \neg P)$. The new rule for negation introduction would be:

$$\frac{\vdash \delta P \quad P \vdash}{\vdash \neg P}$$

As it happens, this is the only required amendment to the basic rules for propositional calculus given by Woodcock and Brien [18], although some new basic rules are required (see Fig. 16). Note that the rule for negation introduction can be derived from the axioms in Fig. 16.

$$\frac{\vdash P}{\vdash \neg\neg P} \qquad \frac{P \vdash}{\neg\neg P \vdash} \qquad \frac{\vdash P}{\neg P \vdash}$$

$$\frac{P, Q \vdash}{P \wedge Q \vdash} \qquad \frac{\vdash P \quad \vdash Q}{\vdash P \wedge Q} \qquad \frac{\vdash \neg P, \neg Q}{\vdash \neg(P \wedge Q)} \qquad \frac{\neg P \vdash \quad \neg Q \vdash}{\neg(P \wedge Q) \vdash}$$

Fig. 16. A W style axiomatization of propositional LPF.

Under a 'strict' interpretation of equality, the formula $a = b$ would denote a truth value iff a and b denote values of the same underlying type: *viz.*

$$a\colon T, b\colon T \;\vdash\; \delta(a = b)$$

where T is a metavariable ranging over base types. From this and Leibniz's rule can be derived a (modified) law of reflection:

$$t\colon T \;\vdash\; t = t$$

8 Conclusions

Z and VDM are very closely related as specification methods, but there are some differences which affect the nature of the analysis techniques they support. This paper has attempted to describe some of the specification verification techniques available to VDM which might not be so familiar to Z users: in particular, the analysis of partial terms and the use of a Logic of Partial Functions to ensure that specifications are mathematically well-formed, and the use of satisfiability analysis to show they are mathematically meaningful.

It is not necessary to use LPF to derive much of the benefit of partiality analysis. Many incompletenesses in a specification can be revealed simply by restricting oneself to partial functions, explicitly noting their domains of definition, and then systematically checking that such functions are applied only to values in their domains. We have found LPF to be a relatively concise logic which differs minimally from classicial logic while recognizing that terms and formulae do not always define values. In practice, reasoning in LPF is almost identical to reasoning in classical logic, except for the abundance of well-formedness hypotheses to be discharged. We have sketched how the W propositional calculus could be adapted to accommodate the principles of LPF, but more work is needed to carry the ideas through to the rest of W.

In terms of practical consequences for the specifier, adoption of LPF would make it necessary, for example, to add sufficiently many conjuncts to schema bodies to ensure that the conjunction denotes a truth value. For example, the schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ T\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
f : A \mathrel{+\!\!\!\!\!\to} B \\
x : A;\ \ y : B \\
\hline
f(x) = y \\
\hline
\end{array}$$

would require an extra conjunct of the form $x \in \mathsf{dom}\, f$ to ensure that $f(x)$ is well-formed.

Underlying logic aside, for the purposes of specification verification the main differences between Z and VDM-SL are that Z offers better support for structuring specifications, while VDM-SL specifications contain more explicit information (most notably, explicit preconditions in functions and operations). Although it is sometimes cumbersome to have to do so, we believe that preconditions are such an important part of any development methodology that they should always be stated explicitly.

We have argued that the above points, in conjunction with the use of finitary data types, can lead to a deeper understanding of the requirements being formalized. Such analysis often reveals assumptions that may otherwise go overlooked until much later in the development. Considering that top-level specification errors might only become apparent after the system has been implemented and is in operation, such analysis techniques can be highly cost-effective.

Acknowledgements

The author gratefully thanks Erik van Keulen, Ian Hayes, Gordon Rose, Kelvin Ross, John Fitzgerald, Jim Grundy, Tony Cant and colleagues at the SVRC for many useful discussions on these topics, and his ex-`mural` colleagues (and especially Cliff Jones) for collaboration on verification techniques for VDM.

References

1. H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
2. J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: a Practitioner's Guide.* FACIT Series. Springer-Verlag, 1994. ISBN no. 3-540-19813-X.
3. S.M. Brien and J.E. Nicholls. Z Base Standard, Version 1.0. Technical Report SRC D–132, Oxford University Programming Research Group, November 1992.
4. British Standards Institute, Working Group IST/5/19. *VDM Specification Language Proto-Standard: Draft*, November 1993.
5. J.H. Cheng. A logic for partial functions. Technical Report UMCS-86-7-1, University of Manchester, Department of Computer Science, 1986.

6. S. Gilmore. *Correctness-Oriented Approaches to Software Development.* PhD thesis, University of Edinburgh, Department of Computer Science, 1991.
7. A. Hall. A response to Florence, Dougal and Zebedee. *FACS Europe*, 1(1):31–32, 1993.
8. I. Hayes. VDM and Z: A comparative case study. *Formal Aspects of Computing*, 4(1):76–99, 1992.
9. I. Hayes, editor. *Specification Case Studies.* Prentice-Hall, second edition, 1993. First Edition published in 1987.
10. I.J. Hayes, C.B. Jones, and J.E. Nicholls. Understanding the differences between VDM and Z. *FACS Europe*, 1(1):7–30, Autumn 1993.
11. W. Hodges. Another semantics for Z. draft preprint, August 1991.
12. C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, New York, second edition, 1990.
13. C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *Mural: A Formal Development Support System.* Springer-Verlag, 1991.
14. C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. Technical Report Logic Group Preprint Series 89, Department of Philosophy, Utrecht University, April 1993.
15. P.A. Lindsay and E. van Keulen. Case studies in the verification of specifications in Z and VDM. Technical Report TR 94-3, Software Verification Research Centre, University of Queensland, March 1994. Available by anonymous ftp from `ftp.cs.uq.edu.au`.
16. B. Monahan and R. Shaw. Model-based specifications. In J.A. McDermid, editor, *Software Engineer's Reference Book*, chapter 21. Butterworth-Heinemann, London, 1991.
17. J.C.P. Woodcock. Calculating properties of Z specifications. *ACM SigSoft Software Engineering Notes*, 14(5):43–54, 1989.
18. J.C.P. Woodcock and S.M. Brien. W: a logic for Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991.* Springer-Verlag, 1992. Proceedings of the Sixth Annual Z User Meeting.
19. J.B. Wordsworth. *Software Development with Z: a Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley, Wokingham, England, 1992.

This article was processed using the LaTeX macro package with LLNCS style