Linking VDM and Z

Jim Woodcock and Leo Freitas
Department of Computer Science
University of York, York YO10 5DD
United Kingdom
{jim,leo}@cs.york.ac.uk

Abstract

The International Grand Challenge in Verified Software is benchmarking current verification technology by conducting a series of experiments, and one such experiment is to build a verified POSIX-compliant flash filestore. An objective of this experiment is to combine different formal methods, and this raises issues about the different logics used. One significant area of difference is in the treatment of undefined expressions, and we show how this difference can be overcome using a unifying theory. This then allows us to use a theorem prover for Z to verify theorems about a data type specified and refined in VDM.

1. Introduction

The International Grand Challenge in Verified Software [9, 12, 10, 2, 14, 20] is a long-term, coordinated programme of research addressing the theoretical, practical, technological, pedagogical, and sociological challenges facing large-scale software verification. The key points of the Grand Challenge’s Research Roadmap are summarised:

- Verification technology is critical to the design, implementation, and maintenance of reliable software. It can reduce the cost of developing, using, and maintaining highly reliable software. Current approaches to software development inspire little confidence—even widely used software contains large numbers of bugs.
- Formal verification technology is advancing rapidly on a number of fronts and is ready for large-scale experiments to measure progress and stimulate further improvements in the performance, robustness, and functionality of this technology.
- The Verified Software Challenge aims to achieve concrete progress through cooperation and competition based on small, medium, and large-scale published benchmarks. The benchmarks, tools, and experimental results are recorded in a Verified Software Repository [2, 22] to coordinate the activities of the Challenge. The enhancement and combination of existing tools and also the invention and exploration of radical new approaches are proposed. The training of software engineers in the advanced use of verification technology is critical to achieving greater software reliability.
- The research thrusts include theory (e.g., mathematics and logic supporting verification and verification tools), methodology (e.g., specification, verification, and correct-by-construction techniques), and tools (e.g., property checkers, theorem provers, and integrated verification environments).
- The research agenda has been driven by realistic experiments, both in the form of small-scale pilot projects that are used to benchmark the technology and identify research challenges, as well as large-scale experiments.
- To be deemed successful, the verification challenge must yield a comprehensive and convincing verification of one or more substantial software artifacts, such as a web server or an operating system kernel.

In the first conference on Verified Software: Theories, Tools, and Experiments, held in Zurich in 2005, Joshi and Holzmann suggested a new pilot project for the grand challenge: the specification, implementation, and verification of the POSIX filestore interface of UNIX [15]. The project involves a small subset of POSIX suitable for flash-memory hardware with strict fault-tolerant requirements to be used by forthcoming NASA missions. This project poses the following research questions:

- Is the mechanical verification of a practical filestore within the state of the art?
- Can high levels of automation be achieved?
- Can useful guarantees be given about the dependability of the system across different hardware platforms?
- How do different theories, tools, and techniques compare, and how can they be used together?
Our contribution in this paper is to the final question. Many of the formal models being used in the pilot project (e.g., [16]) are written in $\text{Z}$ [18, 19, 21]. However, there is a formal specification and refinement of $\text{B}^\prime$-trees in VDM [7], and the question becomes: “Can we combine this formalisation with the models written in $\text{Z}$?” There are three important semantic issues to be addressed in combining VDM with $\text{Z}$: (i) VDM uses separated preconditions and post-conditions; (ii) VDM’s type system includes invariants on types; and (iii) VDM uses a three-valued logic. In this paper, we consider the third issue.

To motivate the problem, we use Cliff Jones’s $\text{subp}$ function [13, p.74], which he has promoted as a puzzle to test any treatment of undefined expressions. We have been developing a unifying theory of undefinedness based on ideas originally due to Mark Saaltink, and we present an informal overview of some of the results of this theory in section 2. In particular, we show how the Z/Eves theorem prover [17] approaches the problem of reasoning about the use of partial functions. We demonstrate what Z/Eves does with $\text{subp}$ in section 4. In section 5, we consider the consistency of the definition that we used. Our treatment of partial functions relies on our knowing their domain of application, and we show how $\text{subp}$’s domain can be calculated formally. We do this calculation in UTP’s theory of designs [11]. In section 6, we show how these ideas can be applied to a real VDM specification. Finally, in section 7, we draw some conclusions.

2. A unifying theory of undefinedness

Although, for any term $t$ in “ordinary” logic, the proposition $t \equiv t$ is true by the axiom of reflection, one cannot assume in a program that the Boolean expression $1/0 \equiv 1/0$ therefore evaluates to true—much more likely is that it does not evaluate to anything at all. This difference between logic and programs must be reconciled, if a method of refinement from specification to program code is to be shown to preserve correctness. This apparently simple problem has been solved in a number of ways, and the resulting specification languages and their logics have some apparently profound differences. A survey of some of the various approaches may be found in [6].

We consider the use of classical logic to reason about facts in a restricted class of logics: monotonic partial logics (monotonicity is explained below). The soundness of these proofs depends on finding a guard to guarantee definedness of expressions. Our approach is as follows.

The theoretical work is in a setting with an explicit undefined value, and we write $a \not\equiv b$ to mean that $a$ is less defined or equal to $b$. This ordering is lifted in the obvious way to a Hoare-preorder $\sqsubseteq_H$ over sets and compound objects. A semantical system defines the admissible denotations for function and predicate symbols, and gives the meaning of equality, negation, disjunction, definite description, and universal quantification. A model over a semantical system specifies a domain of values and the actual denotations of function and predicate symbols. Bound variables are given values by an assignment over a model. A semantics is given to the logical language in terms of a semantical system, a model, and an assignment. We write $\Sigma \models \Phi$ to mean that $\Phi$ is a truth of $\Sigma$ (its semantics evaluates to true for every model of $\Sigma$ and every assignment over that model). A construct $c$ is well-defined in $\Sigma$ if, for every model $M$ over $\Sigma$ and every assignment $\mathcal{M}$, the semantics of $c$ does not evaluate to the undefined value. $G$ is a guard for a construct $c$ in $\Sigma$ provided: (i) $c$ is well-defined in $\Sigma$; and (ii) if $G$ is a truth of $\Sigma$, then $c$ is well-defined.

The Guard Theorem states the following.

Suppose that $\Sigma \subseteq_H \Sigma'$, either $\Sigma$ or $\Sigma'$ is monotonic with respect to $\subseteq$, and that $G$ is a guard for $\Phi$ in $\Sigma$. Then

$$\text{if } \Sigma' \models G \text{ and } \Sigma \models \Phi, \text{ then } \Sigma \models \Phi$$

We show how to use the Guard Theorem to prove results in one semantical system that are then valid in another system.

The SLK Lemma relates three specific monotonic semantical systems. The first is the strict semantical system ($\Sigma_e$), where every function and predicate symbol and every logical operator is strict in each argument (an undefined argument produces an undefined result). The second is Kleene’s three-valued logic ($\Sigma_{kl}$), where everything is monotonic. The third is McCarthy’s left-to-right semantical system ($\Sigma_{lr}$), where function and predicate symbols are monotonic, equality, definite description, universal quantification, and negation are all strict, and disjunction is evaluated with a short-circuit from left to right. The SLK Lemma orders these systems:

$$\Sigma_s \subseteq \Sigma_{lr} \subseteq \Sigma_k$$

A classical semantical system is one that is at least as defined as the strict system, and where every function and predicate symbol is definite (that is, co-strict: undefined results come only from undefined arguments) and every definite description is defined. A semi-classical system is also one that is at least as defined as the strict system, and where equality and every predicate symbol yield only defined results. The Classical and Semi-classical Lemmas state that every semantical system that is at least as defined as the strict system can be refined into both a classical and a semi-classical system.

The implication ordering on semantical systems ($\Sigma \preceq \Sigma'$) holds exactly when, for every formula $\Phi$, ($\Sigma \models \Phi$) $\Rightarrow$ ($\Sigma' \models \Phi$). The Sub-ordering Lemma states

$$\text{if } \Sigma \subseteq \Sigma' \text{ and } \Sigma' \text{ is monotonic, then } \Sigma \preceq \Sigma'$$
The *Semi-classical Extension Lemma* states the following.

\[
\text{if } \Sigma \text{ is a semi-classical system, then } \Sigma_k \preceq \Sigma \text{ and } \\
\Sigma_{lr} \preceq \Sigma
\]

Now we can explain how the Z/Eves theorem prover works. Z/Eves is based on a purely classical logic. Suppose that we want to prove that the formula \( \Phi \) is true in a Z specification with particular function and predicate symbols. We can use the results described in this section to carry out the proof of \( \Phi \) in classical logic, and know that \( \Phi \) is true in Z, a semi-classical setting. The steps are as follows.

1. Construct an appropriate left-to-right system \( \Sigma_{lr} \).
2. Construct a classical semantical system \( \Sigma' \), such that \( \Sigma_{lr} \subseteq \Sigma' \). This is possible by the *Classical Lemma*, since \( \Sigma \subseteq \Sigma_{lr} \) by SLK.
3. Find a guard \( G \) for \( \Phi \) in \( \Sigma_{lr} \).
4. Show that \( \Sigma' \models G \) and \( \Sigma' \models \Phi \). This demonstration is carried out in the classical setting.
5. By the *Guard Theorem*, we have \( \Sigma_{lr} \models G \), since \( \Sigma_{lr} \subseteq \Sigma' \) (from 2). \( \Sigma_0 \) is a guard for \( \Phi \) in \( \Sigma_{lr} \) (from 3).
6. Construct semi-classical \( \Sigma'' \), such that \( \Sigma_{lr} \subseteq \Sigma'' \). This is possible by the *Semi-classical Lemma*, since \( \Sigma_{lr} \subseteq \Sigma_c \) by SLK.
7. By the *Semi-classical Extension Lemma*, \( \Sigma_{lr} \preceq \Sigma'' \), since \( \Sigma'' \) is semi-classical by 6.
8. Thus, \( \Sigma'' \models \Phi \), from 7, from the definition of implication ordering, and from 5.

Every step is fully automated in Z/Eves, except for step 4, which, in general, requires interactive theorem proving using Z/Eves’ classical logic.

### 3. An unexcluded middle approach

In considering the different approaches to the handling of undefinedness, one approach stands out over all others: that involving the suspension of the law of the excluded middle. Although this approach claims an impeccable pedigree, ascending through Kleene, Łukasiewicz, and even up to Boole himself, it is as the logic in VDM [13] that it has come to prominence. This logic is known as LPF, the Logic of Partial Functions. See [1] for a thorough description of LPF, and [3, 4, 5] for detailed descriptions of the whole area of three-valued logics.

A *total* function returns a result for any argument in its domain; a function which is not total is *strictly partial*. Much of mathematics deals with total functions, but in software development partial functions arise quite naturally. In Jones [13], the following simple example is given of partial subtraction:

\[
\text{subp}(i : \mathbb{Z}, j : \mathbb{Z}) \rightarrow \mathbb{Z}
\]

\[
\begin{array}{ccc}
\text{pre} & j & \leq i \\
\text{post} & r & = i - j
\end{array}
\]

This specification is satisfied by the recursive function

\[
\text{subp}(i, j) \equiv \begin{cases} 
0 & \text{if } i = j \\
1 + \text{subp}(i, j + 1) & \text{else}
\end{cases}
\]

A problem arises with terms constructed from \( \text{subp} \) applied to arguments that do not satisfy the precondition, such as in the expression \( \text{subp}(0, 1) \).

The proof obligation for \( \text{subp} \) is

\[
\forall i, j \in \mathbb{Z} : \text{pre-subp}(i, j) \Rightarrow \\
\text{subp}(i, j) \in \mathbb{Z} \wedge \text{post-subp}(i, j, \text{subp}(i, j))
\]

which is to say

\[
\forall i, j \in \mathbb{Z} : j \leq i \Rightarrow \text{subp}(i, j) \in \mathbb{Z} \wedge \text{subp}(i, j) = i - j
\]

So, when the antecedent of this implication is false, the term involving \( \text{subp} \) does not denote an integer. If terms fail to denote, then what meaning is given to the propositions containing them? The answer to this in LPF is to say that the propositions themselves fail to denote a Boolean value. The approach of LPF is to extend the meaning of the logical operators to handle this problem.

As usual, the propositional connectives are given meaning through truth tables, such as the following for conjunction:

<table>
<thead>
<tr>
<th>( \land )</th>
<th>true</th>
<th>*</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>*</td>
<td>false</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

The asterisk marks a “non-value”. This truth table is an extension of that for the classical operator. The other propositional connectives are described by the tables:

<table>
<thead>
<tr>
<th>( \lor )</th>
<th>true</th>
<th>*</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>*</td>
<td>true</td>
<td>*</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \neg )</th>
<th>true</th>
<th>*</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>*</td>
<td>false</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
Jones gives the intuition behind the truth tables in terms of parallel evaluation of the operands. As soon as a result is determined for one operand, it is considered; if the overall result is determined by this single operand, then the program need not complete the evaluation of the other operand. If the evaluation of the operand fails, then it does not cause the overall evaluation to fail. This leads to the natural ordering we introduced above: \( v_1 \preceq v_2 \) means that \( v_1 \) could become \( v_2 \) if evaluated further. The truth tables are the strongest extensions to the classical ones which are monotonically with respect to \( \preceq \).

The law of the excluded middle does not hold in LPF: the proposition \( p \lor \neg p \) is true only if \( p \) denotes a value. More seriously, the deduction theorem does not hold without qualification in LPF, instead there is a requirement that an assumption is defined before it is discharged. Thus, the technique of proof-by-contradiction is not valid in LPF. Furthermore, many of the classical tautologies do not hold in LPF, again for the lack of defininess.

None of the proposals for the semantics and associated proof theories for Z have taken the LPF approach, that of handling undefinedness at the predicate level. In fact, Spivey[18] argues that the three-valued logic approach fits rather better with the VDM way of doing things, than with the Z way. In particular, he observes that in VDM, a predicate is regarded as a Boolean-valued function, and if undefinedness is to be tolerated at the term level, then it is natural to consider an undefined logical value, since a predicate is really just another term. In Z, predicates are regarded more as properties, and a property is represented by the objects that possess that property. Objects are regarded as being included or excluded from sets, with no room for a third possibility.

4. Z/Eves and \( \text{subp} \)

When Z/Eves checks the equation that defines \( \text{subp} \), it generates a domain check, using the guards from \( \Sigma_{lr} \), the left-to-right semantical system. We list exactly what is needed to generate this domain check in Table 1. The domain check proceeds by hand as follows:

\[
( G_{lr}(\text{subp}(i,j)) = (\text{if } i = j \text{ then } 0 \text{ else } 1 + \text{subp}(i,j + 1)))
\]

\[
= \{ \text{equality} \}
G_{lr}(\text{subp}(i,j))
\land G_{lr}((\text{if } i = j \text{ then } 0 \text{ else } 1 + \text{subp}(i,j + 1)))
= \{ \text{function application, conditional} \}
G_{lr}(\text{subp}) \land G_{lr}(i,j) \land (i,j) \in \text{dom subp}
\land G_{lr}(i = j)
\land (\text{if } i = j \text{ then } G_{lr}(0)
\text{ else } G_{lr}(1 + \text{subp}(i,j + 1)))
\]

\[
= \{ \text{identifier, tuple, equality, number, addition} \}
G_{lr}(i) \land G_{lr}(j) \land (i,j) \in \text{dom subp}
\land (\text{if } i = j \text{ then } \true
\text{ else } (G_{lr}(1) \land G_{lr}(\text{subp}(i,j + 1))))
= \{ \text{identifier, number, function application} \}
(i,j) \in \text{dom subp}
\land (\text{if } i = j \text{ then } \true
\text{ else } (G_{lr}(\text{subp}) \land G_{lr}(i,j + 1)
\land (i,j + 1) \in \text{dom subp}))
= \{ \text{identifier, tuple} \}
(i,j) \in \text{dom subp}
\land (\text{if } i = j \text{ then } \true
\text{ else } (G_{lr}(j) \land G_{lr}(1)
\land (i,j + 1) \in \text{dom subp})
= \{ \text{identifier, addition} \}
(i,j) \in \text{dom subp}
\land (\text{if } i = j \text{ then } \true
\text{ else } (G_{lr}(j) \land G_{lr}(1)
\land (i,j + 1) \in \text{dom subp})
= \{ \text{propositional calculus} \}
(i,j) \in \text{dom subp}
\land (i = j \lor (i,j + 1) \in \text{dom subp})
\]

The guard makes sure that every application of \( \text{subp} \) is within its domain, and this applies to the other occurrences of \( \text{subp} \), not just its defining equation. So let us see what Z/Eves does. First, we try a naive definition of \( \text{subp} \):

\[
\forall i,j : \mathbb{Z} \cdot \text{subp}(i,j) = (\text{if } i = j \text{ then } 0
\text{ else } 1 + \text{subp}(i,j + 1))
\]

Z/Eves automatically generates the following domain check

\[
\text{local subp} \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \land i \in \mathbb{Z} \land j \in \mathbb{Z}
\Rightarrow (i,j) \in \text{dom local subp}
\land (\text{if } i = j \text{ then } \true
\text{ else } (i,1+j) \in \text{dom local subp})
\]

which simplifies to

\[
\text{local subp} \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \land i \in \mathbb{Z} \land j \in \mathbb{Z}
\Rightarrow (i,j) \in \text{dom local subp}
\land (i = j \lor (i,1+j) \in \text{dom local subp})
\]

but there is then no possibility of further progress. Let us look at some applied occurrences: here are two global variables defined in terms of \( \text{subp} \).
Z/Eves generates the following domain check.

\[
\text{local good} \in \mathbb{Z} \land \text{local bad} \in \mathbb{Z} \\
\Rightarrow (5, 3) \in \text{dom subp} \\
\land (\text{local good} = \text{subp}(5, 3) \Rightarrow (3, 5) \in \text{dom subp})
\]

Again, the guards guarantee that the application of subp is well defined, and we will not be able to discharge these guards unless we know what the domain actually is, and so we must specify it. It seems fairly clear that \(i\) must be at least as large as \(j\), otherwise the recursion never terminates.

\[
\text{dom subp} = \{ i, j : \mathbb{Z} \mid i \geq j \}
\]

We now add this definition of the domain to the axiomatic definition; a small technicality is to increase the level of automation by announcing the defining equation as a rewrite rule called.domSubp. At the same time, we add the requirements on \(i\) and \(j\).

\[
\text{subp} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \\
\langle \text{rule domSubp} \rangle \text{ dom subp} = \{ i, j : \mathbb{Z} \mid i \geq j \} \\
\forall i, j : \mathbb{Z} \quad (i, j) \in \text{dom subp} \\
\land (i \neq j \Rightarrow (i, j + 1) \in \text{dom subp}) \\
\Rightarrow \text{subp}(i, j) = (\text{if } i = j \text{ then } 0 \\
\text{else } 1 + \text{subp}(i, j + 1))
\]

Z/Eves now generates the following domain check.

\[
\text{local subp} \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \\
\land \text{dom local subp} = \{ i : \mathbb{Z} , j : \mathbb{Z} \mid i \geq j \} \\
\land (i \_0 \in \mathbb{Z} \land j \_0 \in \mathbb{Z}) \\
\land ( (i \_0, j \_0) \in \text{dom local subp} \\
\land (i \_0, j \_0 + 1) \in \text{dom local subp} ) \\
\Rightarrow (i \_0, j \_0) \in \text{dom local subp} \\
\land (\text{if } i \_0 = j \_0 \text{ then true} \\
\text{else } (i \_0, j \_0 + 1) \in \text{dom local subp})
\]

In contrast to the domain check for the naive definition, this domain check is automatically proved true. But what happens to good and bad in this world? Our revised definitions do not leave much room, as we see when we examine the revised domain check:

\[
\text{local good} \in \mathbb{Z} \land \text{local bad} \in \mathbb{Z} \\
\Rightarrow \sim \text{local good} = \text{subp}(5, 3)
\]

It is now apparent that subp is improperly applied in the defining equation for bad, and our only hope for a guard for bad lies in a contradiction in the specification of good.

Here are a few more examples of the use of subp. First, instantiating the axiom of reflection with an improper term; then, instantiating the law of the excluded middle.

\[
\text{theorem subpInreflection} \\
\text{subp}(3, 5) = \text{subp}(3, 5)
\]

\[
\text{theorem subpExcludedMiddle} \\
\text{subp}(3, 5) = 0 \lor \text{subp}(3, 5) \neq 0
\]

These theorems are easily proved by Z/Eves with a single prove command in each case.

In Table 1, one technicality appears in the case where \(f\) has not been defined as a function, yet function application \(f(e)\) is attempted. This happens either when \(f\) is defined as a relation, or more commonly, when \(f\) is itself and expression defined as a relation. For example, for a sequence of pairs of integers, suppose we what to specify that those pairs are functional, and that the first element is always smaller than the second. A naive attempt would be to try the axiomatic definition

\[
\forall i, j : \mathbb{Z} \quad (i, j) \in \text{ran s} \Rightarrow (\text{ran s}) i < j
\]

The resulting guard for the application \((\text{ran s}) i\) is

\[
\text{local s} \in \text{seq} (\mathbb{Z} \times \mathbb{Z}) \land (i \in \mathbb{Z} \land j \in \mathbb{Z}) \\
\land (i, j) \in \text{ran local s} \\
\Rightarrow \text{ran local s applies to i}
\]
The \textit{appliesTo} relation is declared as

$$\text{appliesTo}[X, Y] : (X \leftrightarrow Y) \rightarrow X$$

where the following rewriting rule holds

\textbf{theorem} disabled rule appliesToDef \[ X, Y \]

$$\forall R : X \leftrightarrow Y; x : X \bowtie R \text{ appliesTo } x \Leftrightarrow$$

$$\left( \exists y : Y \mid (x, y) \in R \land \forall y' : Y \mid (x, y') \in R \implies y' = y \right)$$

So, after rewriting the guard above using this rule, and instantiating \( y \) with \( j \) with further rewriting, the goal becomes

$$\text{local } s \in \text{seq}(Z \times Z) \land i \in Z \land j \in Z$$

$$\land (i, j) \in \text{ran local} s \land y' \in Z \land (i, y') \in \text{ran local} s$$

$$\land \neg j = y'$$

$$\Rightarrow (\exists y : Z \mid (i, y) \in \text{ran local} s \land$$

$$\forall y_0' : Z \mid (i, y_0') \in \text{ran local} s \land y = y_0')$$

Now it is clear there is a missing assumption that \( \neg j = y' \)

for all \( i, j', y' \in Z \). This can be solved by pushing this property into the type of \( s \).

5. Consistency

But of course, we should demonstrate that our specification is consistent, and that our specification of the domain of \textit{subp} is actually correct. This proof can be carried out entirely in classical logic using standard techniques. To show this, we recast \textit{subp} as a design in the UTP. Instead of using parameters, our version of \textit{subp} (which we call \textit{subp}$_{\text{y}}$) has three state variables: \( x, y, \) and \( z \). The result of computing \textit{subp}$_{\text{y}}(x, y)$ is left in the variable \( z \). The design \textit{subp}$_{\text{y}}$ initialises \( z \) and then executes \( \mu F \), the weakest fixed-point of the function \( F \).

$$\text{subp}$_{\text{y}}$ \DE$$

\begin{align*}
\{ \mu F \}
\end{align*}

The function \( F(X, Y) \) is a conditional design that tests to see if \( x = y \); if it does, then it terminates (\( \mathbb{P}_D \) is the design \textit{skip}); if \( x \neq y \), then the function behaves as the design \( G(X, Y) \). The two parameters are recursion variables and the notation \( P \mathord{<} b \mathord{>} Q \) is an infix conditional expression.

$$F(X, Y) \DE \{ \mathbb{P}_D < x = y \mathord{<} G(X, Y) \}$$

Finally, \( G(X, Y) \) increments both \( y \) and \( z \), and then makes the recursive call to the design \( X \mathord{<} Y \).

$$G(X, Y) \DE \{ y, z := y + 1, z + 1 ; \ X \mathord{<} Y \}$$

The design \( P \mathord{<} Q \) is a relation with precondition \( P \) and postcondition \( Q \). It is a shorthand for the relation \( ok \land P \Rightarrow ok' \land Q \), which contains two observations. The boolean variable \( ok \) is an observation that the design has started execution, and the boolean variable \( ok' \) is an observation that it has terminated. Designs give us a way of reasoning about possibly non-terminating recursive relations. There is an explicit bottom element \( \bot \), which corresponds to the design \( \text{false} \mathord{<} \text{true} \), which in turn corresponds to the relation \textit{false}.

We now calculate the weakest fixed-point \( \mu F \). We start by transforming \( F \) into a design with explicit precondition and postcondition. First, we do this for \( G \):

$$G(X, Y) \DE \{ \text{definition} \}$$

$$\{ y, z := y + 1, z + 1 ; \ X \mathord{<} Y \}$$

$$\DE \{ \text{design assignment} \}$$

$$\{ \text{true} \mathord{<} y, z := y + 1, z + 1 ; \ X \mathord{<} Y \}$$

$$\DE \{ \text{design sequence} \}$$

$$\{ (y, z := y + 1, z + 1) \mathord{\text{wp}} X \}$$

$$\text{true} \mathord{<} y, z := y + 1, z + 1 ; \ Y \)$$

The relation \( P \mathord{\text{wp}} Q \) is the weakest precondition for the relation \( P \) to achieve the postcondition \( Q \). This derivation uses two assignments: one is a simple relation \( x := e \), which means \( x' = e \). The other is a design \( x := \mathbb{P}_D e \), which means \( \text{true} \mathord{<} x := e \), provided that \( e \) is well defined. Now for \( F \):

$$F(X, Y) \DE \{ \text{definition} \}$$

$$\{ \mathbb{P}_D < x = y \mathord{<} G(X, Y) \}$$

$$\DE \{ \mathbb{P}_D \text{ and } G \}$$

$$\{ \text{true} < y, z := y + 1, z + 1 \mathord{\text{wp}} X \}$$

$$\Rightarrow \{ y, z := y + 1, z + 1 ; \ Y \}$$

$$\DE \{ \text{design conditional} \}$$

$$\{ \text{true} < x := y \mathord{<} \mathbb{P}_D (y, z := y + 1, z + 1 \mathord{\text{wp}} X) \}$$

$$\Rightarrow \{ \mathbb{P}_D < x := y \mathord{<} y, z := y + 1, z + 1 ; \ Y \}$$

$$\DE \{ \text{simplification} \}$$

$$\{ \mathbb{P}_D < x := y \mathord{<} y, z := y + 1, z + 1 \mathord{\text{wp}} X \}$$

$$\Rightarrow \{ \mathbb{P}_D < x := y \mathord{<} y, z := y + 1, z + 1 ; \ Y \}$$

$$\DE \{ \text{introduce definitions} \}$$

$$H(X) \mathord{\text{=<}} I(Y)$$

So we see that \( F(X, Y) \) can be expressed as a design with an explicit precondition \( H(X) \) and an explicit postcondition \( I(Y) \), where

$$H(X) = (\neg x = y \mathord{<} y, z := y + 1, z + 1 \mathord{\text{wp}} X)$$

$$I(Y) = (\mathbb{P}_D < x := y \mathord{<} y, z := y + 1, z + 1 ; \ Y)$$
We want the weakest fixed-point of $F$, and fortunately, the weakest fixed-point of an explicit design can also be expressed as an explicit design (the lattice of designs is closed under recursion).

\[
\begin{align*}
\mu F &= (P \uparrow \bigvee) \\
P &\equiv \nu H \\
Q &\equiv (\mu Y \cdot P \Rightarrow I(Y))
\end{align*}
\]

So here is the explicit precondition for $F$: it is the strongest fixed-point of $H$. Actually, $H$ has a unique fixed-point, a fact that we now prove. First we construct an approximation chain. The $n$-th approximation to $x \geq y$ is $E(n)$, where

\[
E(n) = x \geq y \land x - y < n \\
E \equiv \{ E(i) \mid i \in \mathbb{N} \}
\]

These approximations form a descending chain, with $E(0)$ at the top of the chain:

\[
\begin{align*}
E(0) &= \{ \text{E(n) definition} \} \\
x &\geq y \land x - y < 0 \\
&= \{ \text{arithmetic} \} \\
&\text{false}
\end{align*}
\]

and each element in the chain implying the next:

\[
\begin{align*}
E(i) &= \{ \text{E(i) definition} \} \\
x &\geq y \land x - y < i \\
\Rightarrow &\{ \geq \text{property} \} \\
x &\geq y \land x - y < i + 1 \\
&= \{ \text{E(i+1) definition} \} \\
E(i+1)
\end{align*}
\]

This approximation chain converges on our precondition. Let the convergence condition $\bigvee i \cdot E(i)$ be $C$. Then we can show convergence:

\[
\begin{align*}
\bigvee i \cdot E(i) &= \{ \text{E(i) definition} \} \\
\bigvee i \cdot x &\geq y \land x - y < i \\
&= \{ \text{contract scope} \} \\
x &\geq y \land (\bigvee i \cdot x - y < i) \\
&= \{ \geq \text{property} \} \\
x &\geq y
\end{align*}
\]

Now we show that $H$ is constructive, with respect to the approximation chain $E$.

\[
H(X \land E(n)) \land E(n+1)
\]

\[
= \{ \text{H definition} \} \\
(\neg x = y \Rightarrow (y, z := y + 1, z + 1 \text{ wp } (X \land E(n)))) \\
\land E(n+1)
\]

\[
= \{ \text{wp conjunctivity} \} \\
(\neg x = y \Rightarrow (y, z := y + 1, z + 1 \text{ wp } X)) \\
\land (y, z := y + 1, z + 1 \text{ wp } E(n))) \\
\land E(n+1)
\]

\[
= \{ \text{propositional calculus} \} \\
(\neg x = y \Rightarrow (y, z := y + 1, z + 1 \text{ wp } X)) \\
\land (\neg x = y \Rightarrow (y, z := y + 1, z + 1 \text{ wp } E(n))) \\
\land E(n+1)
\]

\[
= \{ \text{H definition} \} \\
H(X) \\
\land (\neg x = y \Rightarrow (y, z := y + 1, z + 1 \text{ wp } E(n))) \\
\land E(n+1)
\]

\[
= \{ \text{E definition} \} \\
H(X) \\
\land (\neg x = y \Rightarrow (y, z := y + 1, z + 1 \text{ wp } E(n))) \\
\land E(n+1)
\]

\[
= \{ \text{propositional calculus} \} \\
H(X) \\
\land (\neg x = y \Rightarrow x \geq y + 1) \\
\land (\neg x = y \Rightarrow x - y - 1 < n) \\
\land E(n+1)
\]

\[
= \{ \geq \text{property} \} \\
H(X) \\
\land x \geq y \\
\land (\neg x = y \Rightarrow x - y - 1 < n) \\
\land E(n+1)
\]

\[
= \{ \text{prop. calculus } (x = y \Rightarrow x - y - 1 < n) \} \\
H(X) \\
\land x \geq y \\
\land x - y - 1 < n \\
\land E(n+1)
\]

\[
= \{ \text{E definition} \} \\
H(X) \\
\land x \geq y \\
\land x - y < n + 1 \\
\land x \geq y \land x - y < n + 1
\]

\[
= \{ \text{propositional calculus} \}
\]
\[ H(X) \]
\[ \land x \geq y \land x - y < n + 1 \]
\[ = \{ E \text{ definition } \} \]
\[ H(X) \land E(n + 1) \]

Since \( E \) is an approximation chain for \( C \) and \( H \) is \( E \)-constructive, then we have that
\[ C \land \mu H = C \land \nu H \]

That is, the weakest and strongest fixed-points are equal, modulo \( C \). But \( C \) is also a fixed point.
\[ H(C) \]
\[ = \{ H \text{ definition } \} \]
\[ \neg x = y \Rightarrow (y, z := y + 1, z + 1 \text{ wp } C) \]
\[ = \{ C \text{ definition } \} \]
\[ \neg x = y \Rightarrow (y, z := y + 1, z + 1 \text{ wp } x \geq y) \]
\[ = \{ \text{ wp assignment } \} \]
\[ \neg x = y \Rightarrow x \geq y + 1 \]
\[ = \{ \text{ propositional calculus } \} \]
\[ x = y \lor x \geq y + 1 \]
\[ = \{ \text{ arithmetic } \} \]
\[ x \geq y \]
\[ = \{ C \text{ definition } \} \]
\[ C \]

The last two facts mean that the strongest fixed-point is actually the fixed-point \( C \).
\[ C \land \mu H = C \land \nu H \]
\[ = \{ \text{ predicate calculus } \} \]
\[ [C \Rightarrow (\mu H = \nu H)] \]
\[ = \{ \text{ fixed point } C \} \]
\[ [C \Rightarrow (\nu H \subseteq C)] \]
\[ \Rightarrow \{ \text{ Leibniz } \} \]
\[ [C \Rightarrow (\nu H \subseteq C)] \]
\[ \Rightarrow \{ \text{ refinement } \} \]
\[ [C \Rightarrow [C \Rightarrow \nu H]] \]
\[ \Rightarrow \{ \text{ predicate calculus } \} \]
\[ [C \Rightarrow \nu H] \]
\[ = \{ \text{ refinement } \} \]
\[ \nu H \subseteq C \]
\[ = \{ \nu H \text{ is strongest fixed-point } \} \]
\[ \nu H = C \]

We could continue and calculate the postcondition \( Q \), so we would have a representation of \( subp \) as an explicit precondition/postcondition pair. But this is not necessary for the current discussion: we have proved that our specification of the domain of \( subp \) is correct.

### 6. Verifying VDM with Z/Eves

In this section we show the VDM specification of an abstract mapping and the data structure used to implement it.

Quoting from [7]:

Since it is intended to use a binary tree as a structure in which to store a mapping from keys to data, using a mapping as the data type in the initial specification is a good abstraction. It allows the find operation to be specified in terms of its effect without prescribing how it is to work.

\[ S0 = \text{ Key } \rightarrow \text{ Data} \]

\[ \text{ APPLY0} \]

\[ \text{ States } S0 \]

\[ \text{ Type } : \text{ Key } \rightarrow \text{ Data} \]

\[ \text{ pre-APPLY0}(s_0, k) \equiv k \in \text{ dom } s_0 \]

\[ \text{ post-APPLY0}(s_0, k, s'_0, r) \]

\[ \equiv s'_0 = s_0 \text{ and } r = s_0(k) \]

This stage of the refinement represents a mapping as a binary tree. If the tree is not empty, each node of the tree contains a key and its associated data as well as a pair of pointers (either or both of which may be null), to a left and right subtree respectively. All the keys occurring in the left subtree of a particular node will have values less than the value of the key in the node, and the values of the keys of the right subtree will be greater than the node key. This is stated in the data type invariant.

\[ \text{ S1 } = [\text{ Bt1}] \]

\[ \text{ Bt1 } = \text{ S1 Key Data S1} \]

\[ \text{ invS1 } : \text{ S1 } \rightarrow \text{ Bool} \]

\[ \text{ invS1}(s_1) \equiv \]

\[ \text{ s1 = nil or} \]

\[ \text{ let } \langle l, k, d, r \rangle = s_1 \text{ in} \]

\[ \langle \forall k \in \text{xks}(l) \rangle(k < k) \text{ and invS1}(l) \text{ and} \]

\[ \langle \forall k \in \text{xks}(r) \rangle(r < k) \text{ and invS1}(r) \]

\[ \text{ where} \]

\[ \text{xks : S1 } \rightarrow \text{ Key-set} \]

\[ \text{xks}(s_1) \equiv \]

\[ \text{ if s1 = nil then } \}

\[ \text{ else let } \langle l, k, d, r \rangle = s_1 \text{ in} \]

\[ \{ k \} \cup \text{ union } \{ \text{xks}(l), \text{xks}(r) \} \]
We translate the VDM specification into the dialect of Z used by Z/Eves. We introduce keys as natural numbers, \( \text{Data} \) as a given set, and the type of Boolean values. The latter is necessary because we want to have Boolean-valued expressions.

\[
\begin{align*}
\text{Key} &::= \mathbb{N} \\
[\text{Data}] &::= \text{TRUE} \mid \text{FALSE}
\end{align*}
\]

Now we can define the type of the abstract mapping by translating VDM’s mapping constructor into Z’s finite (partial) function constructor.

\[
\begin{align*}
S_\cdot.0 &::= \text{Key} \Rightarrow [s_\cdot.0 : S_\cdot.0] \\
\text{StateS}_\cdot.0 &::= [s_\cdot.0 : S_\cdot.0]
\end{align*}
\]

The abstract state is modelled using a schema. Now we can specify the abstract apply operation. In order to preserve VDM’s separation of pre- and postconditions, we borrow from UTP the trick of using the observational variables \( \text{ok} \) and \( \text{ok'} \) [11].

\[
\begin{align*}
\text{Apply}_\cdot.0 &::= \\
\Delta \text{StateS}_\cdot.0 &::= \\
\text{key} : \text{Key} ; \text{rl} : \text{Data} &::= \\
\text{ok} , \text{ok'} : \text{Boolean} &::= \\
\text{ok} = \text{TRUE} \land \text{ok'} &::= \text{ok} \land \text{ok'}
\end{align*}
\]

This definition generates a domain check, since it contains a partial function application. The domain check is discharged completely automatically.

We can now prove VDM’s applicability theorem, which demonstrates that the asserted precondition is strong enough to guarantee the actual precondition required by the postcondition. This theorem is easily stated using Z’s pre operator, which extract the exact precondition.

\[
\begin{align*}
\text{theorem PreApply}_\cdot.0 &::= \\
\forall \text{StateS}_\cdot.0 ; \{ \text{key} : \text{Key} \mid \text{ok} = \text{TRUE} \land \\
\text{key} \in \text{dom} s_\cdot.0 \} ; \text{pre Apply}_\cdot.0
\end{align*}
\]

This theorem is discharged automatically. Next, the VDM abstract syntax is translated into a Z free type.

\[
T_\cdot.1 ::= \text{nil} \mid \text{node}\langle T_\cdot.1 \times \text{Key} \times \text{Data} \times T_\cdot.1 \rangle
\]

Finally, we need to define the type \( S_1 \) with its invariant. We do this by first introducing the recursive function \( \text{xks} \) that collects the keys in a tree.

\[
xks : T_\cdot.1 \rightarrow \mathbb{P} \text{Key}
\]

\[
\begin{align*}
\forall s_\cdot.1 : T_\cdot.1 \bullet \\
xks s_\cdot.1 &= \\
\text{if } s_\cdot.1 = \text{nil} \text{ then } 0 \\
\text{else } \{ \text{let } \text{left} = (\text{node}^\sim s_\cdot.1).1; \\
\text{key} = (\text{node}^\sim s_\cdot.1).2; \\
\text{right} = (\text{node}^\sim s_\cdot.1).4 \bullet \\
\{ \text{key} \} \cup \bigcup \{(\text{xks left}, (\text{xks right}) \}
\end{align*}
\]

This definition generates domain checks for the six partial function applications present. Notice the use of the inverse free type constructor \( \text{node}^\sim \). Discharging the domain checks requires us to know that \( \text{node}^\sim \) is a function at the point \( s_\cdot.1 \). Fortunately, we already have a theory of free types that allows us to automate this check. This theory was originally developed for the Mondex case study [23, 8]. Now for the definition of \( S_1 \).

\[
\begin{align*}
S_\cdot.1 &::= \mathbb{P} T_\cdot.1 \\
\forall s_\cdot.1 : T_\cdot.1 \bullet \\
s_\cdot.1 \in S_\cdot.1 &::= \\
\text{if } s_\cdot.1 = \text{nil} \text{ then } 0 \\
\text{else } \{ \text{let } \text{left} = (\text{node}^\sim s_\cdot.1).1; \\
\text{key} = (\text{node}^\sim s_\cdot.1).2; \\
\text{right} = (\text{node}^\sim s_\cdot.1).4 \bullet \\
\{ \text{key} \} \cup \bigcup \{(\text{xks left}, (\text{xks right}) \}
\end{align*}
\]

The domain checks here are very similar to those for \( \text{xks} \), and again they are polished off automatically.

The development proceeds like this, translating the VDM definitions into Z, and proving the domain checks as we go. The result is a Z specification and refinement that does not abuse the notion of undefinedness, and so has the same properties as the LPF-based VDM specification. Of course, correctness also depends on our informal translation of VDM’s syntax into that of Z, but we have justified that the two treatments of undefinedness are strictly compatible for this development.

### 7. Conclusions

In this paper, we discussed the problem of linking VDM and Z, focusing on the problem of their different treatments of undefined values. We gave an informal description of a unifying theory of undefined values, which is currently under construction. We showed how this theory allows us to use the Z/Eves theorem prover to prove facts about a VDM specification.

We presented a very small fragment of the VDM development of an implementation of an abstract mapping. A
second, much larger development using B⁺-trees is of more interest, and we are working on its formalisation in Z/Eves. Further work will include formalising the two-way translation between VDM and Z. Our experience has shown that we need to develop theories to automate reasoning about mathematical data types. A library of such theories is just one outcome from the Grand Challenge experiments.

Acknowledgements

We are grateful to the following people for comments on the work reported in this paper: Andrew Butterfield, Ana Cavalcanti, John Fitzgerald, and Cliff Jones. The ideas were aired in 2007 at the IFM conference at Oxford, the Z User Meeting in New Zealand, the IFIP WG 2.3 meeting in Sydney, and a RefineNet meeting at the University of Essex, at seminars in Newcastle, Pernambuco, and York, and an impromptu talk at a Summer School in Shanghai in honour of the 70th birthdays of Dines Bjørner and Zhou Chaochen.

References